Ex libris
universitatis
albertaensis

Digitized by the Internet Archive
in 2026 with funding from
University of Alberta Library

https://archive.org/details/0162001340676

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: CHUNG HEE HWANG

TITLE OF THESIS: FILE ORGANIZATION SCHEMES FOR GEOMETRIC DATA

DEGREE FOR WHICH THESIS WAS PRESENTED: MASTER OF SCIENCE

YEAR THIS DEGREE GRANTED: FALL 1985

Permission is hereby granted to THE UNIVERSITY OF ALBERTA LIBRARY to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

THE UNIVERSITY OF ALBERTA

FILE ORGANIZATION SCHEMES FOR GEOMETRIC DATA

by

CHUNG HEE HWANG

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE

OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTING SCIENCE

EDMONTON, ALBERTA

FALL 1985

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled FILE ORGANIZATION SCHEMES FOR GEOMETRIC DATA submitted by CHUNG HEE HWANG in partial fulfilment of the requirements for the degree of MASTER OF SCIENCE.

Abstract

A file organization scheme is proposed for representing a polygon network of geometric data by a quadtree, and algorithms for typical spatial queries are described. The proposed scheme is an adaptive cell method, and it has been developed based on *extendible hashing* and *interpolation-based index maintenance*. It aims to locate the record associated with a given key with one disk access or at most two, maintaining a high storage utilization ratio. Its advantages over other file structure proposed for large quadtrees, namely, the B-tree, are that the proposed scheme retrieves a record with $O(1)$, not $O(\log n)$, disk accesses, where n is the number of records in the file, and that it efficiently handles polygon union and intersection and range search. Experimental results with a set of real data are encouraging.

Acknowledgement

I wish to thank my mother, Mrs. S. J. Hwang, for her love and encouragement without which I would not have completed this work.

With respect to the content and presentation of the thesis, I would like to thank Dr. Wayne A. Davis, my supervisor, for his invaluable guidance and assistance. I also appreciate the helpful comments provided by the members of my examination committee, Dr. Stanley Cabay, Dr. Mark Green and Dr. J. Ronald Eyton. Special thanks are due to Dr. Cabay for his constructive criticisms and insightful discussions that helped me a lot during the course of this research.

I gratefully acknowledge the financial support of the Department of Computing Science and the Faculty of Graduate Studies and Research of the University of Alberta and the Natural Sciences and Engineering Research Council of Canada in the form of research assistantships.

Table of Contents

Chapter Page

1. INTRODUCTION 1

2. THE POLYGON NETWORK: A CONCEPTUAL FRAMEWORK 7

2.1 Introduction 7

2.2 Spatial Queries on a Polygon Network 10

2.3 Data Structures 13

2.3.1 The Region Quadtree 16

2.4 File Structures 22

3. DYNAMIC HASHING AND CELL METHODS 26

3.1 Introduction 26

3.2 Extendible Hashing 29

3.3 Linear Hashing 33

3.4 Cell Methods Based on Dynamic Hashing 39

4. A PROPOSED FILE ORGANIZATION SCHEME 47

4.1 Definitions and Notations 47

4.2 Mapping between Regions in the Study Area and Data Buckets 53

4.2.1 Characteristics of Data Buckets 54

4.2.2 Correspondence between Blocks and Data Buckets 56

4.2.3 Correspondence between Subblocks and Data Buckets 60

4.3 Provisions for Dynamic File Maintenance 62

4.3.1 Splitting 62

4.3.2 Merging 70

4.4 Support for Spatial Queries 74

4.4.1 Point-in-Polgyon 74

4.4.2 Set Operations 76

4.4.3 Range Search 78

5. EVALUATION OF THE PROPOSED SCHEME 82

5.1 Types of Spatial Queries Supported and Their Efficiency 82

5.2 Access Efficiency and Storage Utilization 84

5.2.1 Experimental Results 86

6. CONCLUDING REMARKS 94

BIBLIOGRAPHY 98

APPENDIX 102

A. Data Structures 102

B. Algorithm Access 104

C. Algorithm Insert 106

D. Algorithm Delete 109

E. Algorithm SequenRetrieve 112

F. Algorithm RangeSearch 114

List of Tables

Table Page

1. Experimental Results for the Proposed File Scheme with an Upper Limit of the Load Factor 0.65 90

2. Experimental Results for the Proposed File Scheme with an Upper Limit of the Load Factor 0.70 91

3. Summary of Experimental Results 93

List of Figures

Figure		Page
2.1	Polygons and a Polygon Network	8
2.2	Examples of a Polygon Network	9
2.3	polygon Overlay	12
2.4	A Polygon Network and Its Representations	18
3.1	Extendible Hashing	32
3.2	Linear Hashing	38
3.3	Comparison of Search Space Partitions	44
4.1	1-Dimensional Transformation of Polygon Network Data	49
4.2	Partition of Data Space	55
4.3	Correspondence between Directory Elements and Cells in the Study Area	57
4.4	Correspondence between Blocks and Data Buckets	59
4.5	Subblocks and their Indexes	61
4.6	Correspondence between Subblocks and Data Buckets	62
4.7a	Dynamic File Maintenance: Initial State (a)	65
4.7b	Dynamic File Maintenance: Splitting (b)	66
4.7c	Dynamic File Maintenance: Splitting (c)	67
4.7d	Dynamic File Maintenance: Splitting (d)	67
4.7e	Dynamic File Maintenance: Splitting (e)	68
4.7f	Dynamic File Maintenance: Splitting (f)	68
4.7g	Dynamic File Maintenance: Splitting (g)	69
4.8a	Dynamic File Maintenance: Merging (a)	72
4.8b	Dynamic File Maintenance: Merging (b)	73
4.8c	Dynamic File Maintenance: Merging (c)	73

4.9 Point-in-polygon 75
4.10 Set Operations 77
4.11 Range Search 79
5.1 A Surficial Geology Map 87

CHAPTER 1

INTRODUCTION

This thesis is concerned with file schemes for organizing geometric data representing a polygon network. Polygon networks have been an interesting research area in the domain of computational geometry for some time. Recently, however, they have received increased attention owing to their applications in various fields such as computer graphics, scene analysis, geometric modeling, computer aided design and geographic information systems.

The problem of how to organize data in external memory so as to facilitate subsequent retrieval of the data is crucial in most large databases where the file size precludes the possibility of keeping all the data in main memory. In geometric databases, however, this problem is even more complicated due to the spatial nature of the queries that should be handled. Spatial queries are inherently different from ordinary data processing queries in that retrieval by coordinates is usually required. Thus, a special file organization scheme is desired for geometric data that allows spatial queries to be processed efficiently.

The performance of geometric databases is assessed typically by measuring the average response time for each type of query. One of the major components of response time

is access time, which in turn is best measured in terms of the number of disk accesses involved. This is due to the fact that the time to access data in external memory usually involves milliseconds or tens of milliseconds, while the time to access data in main memory is usually expressed in hundreds of nanoseconds. Thus, the cost of most spatial queries is I/O bound rather than CPU bound. Consequently, minimizing the number of disk accesses is one of the most important goals of an efficient file scheme.

To minimize the number of disk accesses, it should be ensured that only those parts of a file that are spatially relevant to the processing of a given query are retrieved. In this regard, an efficient file scheme is required to have a "localization" or "windowing" property. A familiar file organization technique with a localization property is a cell method (sometimes called a grid method) which is basically a coordinate based access technique [18,pp.554-555].

Another important aspect required of an efficient file scheme is dynamic data maintenance. A file scheme is called "dynamic" if the file structure can adapt its shape automatically in response to the external demands such as insertion or deletion of records. The rationale for this requirement is to maintain a uniform access time over the entire file and to keep the storage space as small as possible.

Much research has been devoted to applying cell methods to point data in a k-dimensional space (for a survey, see [4] and [28,pp.239-244]). Some of the schemes are dynamic, others are not. On the other hand, for line or polygon network data, especially those encoded in raster format, research has not been extensive.

Traditionally, polygon network data have been represented in vector format. Although vector representation has advantages of being more flexible and easy to extract boundary information, raster representation of a polygon network is becoming popular because of the simplicity in handling many spatial queries, e.g., point-in-polygon and set operations. In particular, quadtree encoding of a polygon network has received increased attention. The quadtree is a dynamic data structure developed to reduce the storage requirement of raster representation by aggregating homogeneous cells. Nevertheless, as the original quadtree concept was based on the assumption that quadtrees were resident in main memory, quadtree structures may not be directly applicable to the data resident in external memory. For example, the need to follow pointers may lead to a larger number of page faults than are acceptable in an interactive environment.

In an effort to overcome the frequent page fault problem, there have been studies to represent a quadtree as a linear quadtree and use a B-tree file structure in

organizing the data [1,2,27,29]. While the B-tree organization of quadtree encoded data is a significant improvement over the original quadtree organization in the number of disk accesses required to retrieve a single record in a file, the absence of a localization property is its primary disadvantage. That is, a query usually requires the whole file to be retrieved even though the query can be answered with information from a small part of the file. For example, range search is very awkward and set operations are not efficient because there is no implied connection between data buckets in physical storage and regions in the search space. Furthermore, the B-tree organization of quadtree encoded data still needs several disk accesses to retrieve each record because it is essentially a tree that is accessible with $O(\log n)$ I/O operations, where n is the number of records in the file.

From this perspective, this thesis intends to develop a file organization scheme for polygon network data encoded as a quadtree with the following features: (1) it should have a localization property, (2) it should be a dynamic scheme, (3) it should locate the record associated with a given key with one disk access or at most two, and (4) the storage requirement should be $O(n)$, where n is the number of records in the file, with a high storage utilization ratio, e.g., higher than 60%.

The rest of the thesis is organized as follows. Chapter 2 discusses preliminary concepts of a polygon network. It introduces various examples of a polygon network commonly encountered in practice and defines typical spatial queries performed on a polygon network. Of particular interest are the point-in-polygon problem, set operations and range search. In addition, it deals with representational problems, i.e., vector representation and raster representation of a polygon network are discussed with their relative merits. Also addressed are the considerations needed in designing a file scheme for geometric data in general.

Chapter 3 is about dynamic hashing and its use in implementing a cell method. Various file organization schemes are reviewed with an emphasis on extendible hashing and linear hashing. These hashing schemes lend themselves nicely to cell methods. Moreover, they are dynamic in the sense that they resolve a collision without reorganizing the whole file and yet without deteriorating the performance in access time. The chapter also compares various cell methods that are developed based on dynamic hashing, e.g., the EXCELL method, the grid file and interpolation-based index maintenance.

Chapter 4 proposes a file organization scheme for representing a polygon network by a quadtree. The scheme is an adaptive cell method, and it is developed based on exten-

dible hashing and interpolation-based index maintenance. The main features of the proposed scheme, i.e., the mapping between cells and data buckets and the dynamic nature of its file maintenance, are discussed. Also described is how the proposed scheme supports spatial queries, particularly, range search.

Chapter 5 is devoted to the evaluation of the proposed file scheme. Due to the lack of standardization in measuring the performance of a file scheme representing a polygon network, experimental results of the scheme with a set of real data is presented instead of a theoretical analysis. The results are then compared with those of other file schemes, i.e., the B-tree organization and the EXCELL method.

Concluding remarks and a proposal for future research are presented in Chapter 6.

CHAPTER 2

THE POLYGON NETWORK: A CONCEPTUAL FRAMEWORK

The purpose of this chapter is to give an overview of the polygon network. A polygon network is first defined. Then, various examples of polygon networks commonly encountered are introduced followed by definitions of typical spatial queries performed on a polygon network. Finally, the problems of how to store a polygon network in a computer, i.e., what kind of spatial data structures may be used and what should be considered in designing a file structure, are discussed.

2.1 Introduction

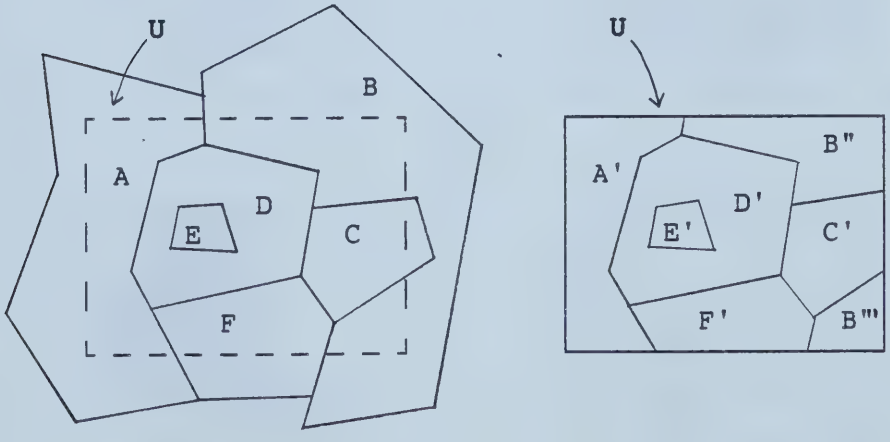
Given a sequence of k points $P_i=(x_i,y_i)$, for $1 \leq i \leq k$, in the plane, a polygon with vertices P_i is the sequence of line segments, called edges, $P_1P_2,P_2P_3,\ldots,P_kP_1$. If these k edges do not have any intersection points, the polygon is simple. A simple polygon divides the plane into two distinct regions. The interior of a simple polygon A is called a *polygonal region A*.

A *complex polygonal region* is a polygonal region which is allowed to have one or more holes in it. For instance, suppose A and B are polygonal regions and B is contained in A. Then, $C = A - B$ is a complex polygonal region. Hereafter, a complex polygonal region is called a *polygon* for

short.

A *polygon network* for a study area is a set of disjoint polygons overlapping the study area such that the set of polygons yields a total partition of the study area. Equivalently, a polygon network can be regarded as a many-to-one mapping between the set of points in the study area and the set of polygons that totally partitions the study area. Each polygon in a polygon network has a unique name. Thus, every edge in a polygon network has associated with it two distinct regions: one on its left, the other on its right.

As an example, consider the polygons A,B,C,D,E and F and the study area **U** in Figure 2.1a. Let $A' = A \cap \mathbf{U}$, $B' = B \cap \mathbf{U}$, ..., $F' = F \cap \mathbf{U}$. Then, each of A',C',D',E' and F' is a polygon, but B' is not. Now, suppose B'' and B''' are non-overlapping polygons and $B'' \cup B''' = B'$ as shown in Figure

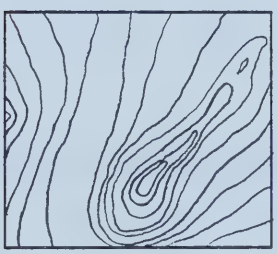


(a) (b)

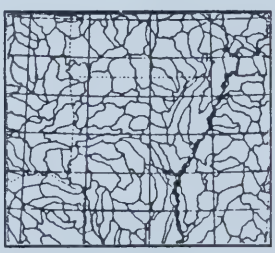
Figure 2.1. Polygons and a Polygon Network

2.1b. Then, $N = \{A', B'', B''', C', D', E', F'\}$ is a polygon network on the study area **U**.

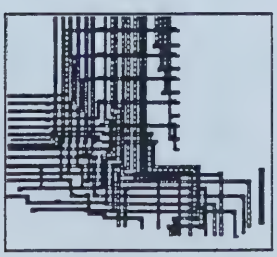
Figure 2.2 shows examples of polygon networks that are commonly encountered in practice. Figures 2.2a and 2.2b are examples from geographic information systems, and Figure 2.2c is from computer aided design. Figure 2.2c is in fact a polygon network of a special type where the background polygon does not have any meaningful information and can thus be ignored in further processing. Figure 2.2d shows a 3-dimensional polygon network, or more precisely, a polyhedron network. Note that the study area in Figure 2.2d is a 3-dimensional hyper-rectangle rather than a 2-dimensional



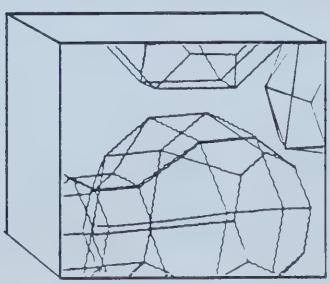
(a) A contour map



(b) A river basin map



(c) Chip configuration



(d) Solid representation

Figure 2.2. Examples of a Polygon Network

rectangle and is partitioned into a set of polyhedrons. This kind of 3-dimensional polygon network can be found in computer graphics and solid modeling.

Although a polygon network can be generalized as a *k*-dimensional network like the one in Figure 2.2d, this thesis deals with only 2-dimensional polygon networks. The schemes developed in this thesis for a 2-dimensional polygon network can be applied to a *k*-dimensional polygon network, but this extension may not be trivial.

2.2 Spatial Queries on a Polygon Network

Typical spatial queries performed on a polygon network are the point-in-polygon query, range search and set operations such as polygon union or intersection and polygon overlay.

A. Point-in-Polygon Problem

Given a polygon network on the study area **U**, a polygon P_o in the network, and a point $(x_o,y_o) \in \mathbf{U}$, the point-in-polygon problem is to determine whether or not the point is contained in P_o. A generalized version of this problem is, given a polygon network on the study area **U** and a point $(x_o,y_o) \in \mathbf{U}$, to determine which polygon contains the point and is termed the *point-in-polygon-network* problem.

B. Range Search

(1) Circular Range Search

Given a polygon network on the study area **U**, a point (x_0,y_0), and the radius r, circular range search is to retrieve information for every point (x,y), where $(x,y) \in \mathbf{U}$ and $(x_0 - x)^2 + (y_0 - y)^2 \leq r^2$.

(2) Orthogonal Range Search

Given a polygon network on the study area **U** and two points (x_1,y_1) and (x_2,y_2) such that $x_1 \leq x_2$ and $y_1 \leq y_2$, orthogonal range search is to retrieve information for every point (x,y), where $(x,y) \in \mathbf{U}$ and $x_1 \leq x \leq x_2$ and $y_1 \leq y \leq y_2$. This operation is analogous to *windowing* in graphics.

C. Set operations

(1) Polygon Union

Given a polygon network on the study area **U** and two adjacent polygons P_1 and P_2 in the network, the polygon union operation obtains a polygon P_3 such that $P_3 = P_1 \cup P_2$. This operation is often called a *polygon merge*.

The polygon union operation can be applied to a pair of polygons from different networks, as well. Given two polygon networks N_1 and N_2 on the same study area **U** and two polygons P_1 and P_2 in the networks N_1 and N_2, respectively, the polygon union operation obtains a polygon P_3 such that $P_3 = P_1 \cup P_2$.

(2) Polygon Intersection

Given two polygon networks N_1 and N_2 on the same study area **U** and two polygons P_1 and P_2 in the networks N_1 and N_2, respectively, the polygon intersection operation obtains a polygon P_3 such that $P_3 = P_1 \cap P_2$.

(3) Polygon Overlay

Polygon overlay is equivalent to polygon intersection except that the operation may be applied to every possible pair of polygons, one from each given network. That is, given two polygon networks N_1 and N_2 on the same study area **U**, the polygon overlay operation is to produce a polygon network N_3 such that $N_3 = N_1 \times N_2$. The resulting network N_3 is a many-to-one mapping from the set of points on **U** to the cartesian product of the set of polygons in N_1 and the set of polygons in N_2. Thus, N_3 associates every point in **U** to two polygons, one in N_1 and one in N_2, as shown in Figure 2.3. This operation is also called *polygon superimposition*.

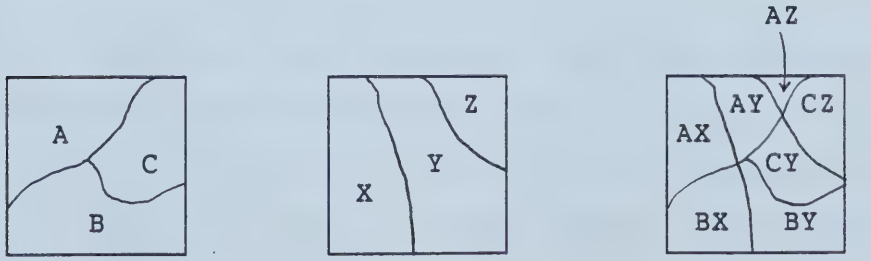


Figure 2.3. Polygon Overlay

In addition to these, another group of operations frequently performed on polygon networks includes geometric calculations such as area and perimeter computing, geometric transformations such as translation and scaling, finding neighbours[1] of a polygon, and displaying the entire polygon network or a part thereof. In this thesis, however, only point-in-polygon, range search and set operations are of interest.

2.3 Data Structures

Two major data structures used to represent a polygon network are *vector* and *raster*. The basic logical entity in vector data structures is a *line*[2] that constitutes the boundaries of polygons. For instance, a polygon is described by an ordered set of lines which in turn are defined by an ordered set of points. In contrast, in raster data structures, the basic logical entity is a *unit cell* that is obtained by subdividing the study area into a rectangular grid.

Both of these data structures have advantages and disadvantages, and the decision of which structure to choose is application dependent. Since a polygon network in raster

---

[1] Two polygons that share a common boundary are *neighbours* to each other.

[2] In this thesis, a *line* is considered to have the following properties:
- it must be continuous,
- it may not necessarily be straight,
- it may not branch, and
- it may not continue through intersections.

representation is described by the interiors of polygons rather than by their boundaries, boundary information is difficult to extract from this representation. Also, queries regarding the topology of the network such as an adjacency test of two polygons require a considerable amount of work in raster representation as there is no easy way of explicitly recording the topological information. In vector representation, however, the boundary or adjacency problem is easily solved because the topology of the network can be recorded in an explicit manner. A common way of recording the topology in vector representation is to attach to each (directed) line the names of polygons separated by the line, for example, LEFT(LineA) = PolygonX and RIGHT(LineA) = PolygonY.

The primary drawback of raster data structures is their wasteful use of computer storage due to their tendency of storing redundant data. Note that the resolution of the grid lines underlying the raster is normally set equal to the highest coordinate resolution required.

On the other hand, vector data structures have algorithmic disadvantages in solving some of the most common problems. For instance, a point-in-polygon test in vector representation ([24] provides a survey on this) requires a non-trivial algorithm (the well-known plumb-line algorithm is of $O(n)$, where n is the number of vertices in the polygon), while the problem in raster representation is

merely that of *lookup* with no processing overhead.

The complexity of performing set operations in vector representation is even worse. Most algorithms developed for polygon intersection using vector data structures involve searching for line segments that are likely to intersect, followed by tests for intersection and calculation of the intersection point, if any. Consequently, these algorithms are not conceptually simple and often require complicated data structures. Therefore, vector representation may become very bad when the polygons involved have a large number of vertices, which is usually the case. In contrast, raster algorithms for polygon intersection are trivial; they are essentially *lookup and compare* operations.

As for polygon overlay in vector representation, the best known algorithm by Nievergelt and Preparata [23] is of $O(n \log n + s \log n)$, where n is the total number of vertices of polygons in the networks and s is the number of intersections of all line segments, with storage overhead of $O(n+s)$. In raster representation, polygon overlay is a problem of $O(n)$, where n is the number of grid cells in the study area, with no storage overhead. Obviously, the different nature of each representation makes it difficult to compare the complexities, but it should be stressed that the latter is algorithmically trivial, while the former is considerably complicated in terms of algorithm development and data structures.

Although vector data structures have been traditionally used to represent geometric features, much interest has been recently given to raster data structures because of their conceptual clarity and ease of manipulation. Also, raster representation is compatible with modern graphic devices. Hence, a wide variety of data compression techniques have been proposed in an effort to avoid the redundancy of the data in raster representation.

Among others, but of particular interest is the *quadtree* which has been developed to reduce the storage requirement by aggregating homogenous cells, i.e., cells contained in the same polygon. One of the advantages of the quadtree is its dynamic structure that allows successive refinement for higher resolution at lower levels in the tree. Furthermore, its ability to perform the point-in-polygon test and set operations quickly has made the quadtree an attractive choice over other alternative structures. Since the quadtree data structure is utilized in this thesis as a means of representing a polygon network, the following section briefly reviews the quadtree. Readers interested in other types of spatial data structures are referred to [9,25].

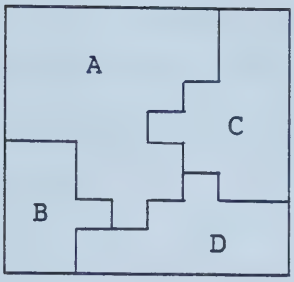
2.3.1 The Region Quadtree

The term quadtree is used to describe 'a class of hierarchical data structures whose common property is that they

are based on the principle of recursive decomposition of space' [28]. Although quadtrees can be used for point data, regions, curves, surfaces and volumes, only those representing 2-dimensional regions will be described in the following discussion. Moreover, the discussion deals with the traditional region quadtree only. However, it should be noted that there have been studies to represent regions using line quadtrees as well. For a more detailed and extensive study on quadtrees, see the comprehensive survey on quadtrees by Samet [28].

The region quadtree (hereafter, quadtree) represents a polygon as a collection of maximal square blocks that partitions the polygon. The scheme is based on the recursive decomposition of the study area into four equal sized quadrants. If a quadrant is not contained entirely in a particular polygon, it is further subdivided into quadrants until every block consists of homogeneous cells. Thus, each block is either entirely contained in a polygon or entirely outside of it. As a result, quadtree blocks have standard locations and standard sizes, i.e., every block is a square whose sides have a length equal to a power of two.

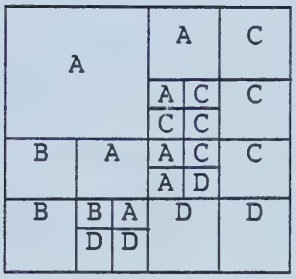
As an example, consider the polygon network shown in Figure 2.4a, which is represented by an image of size $2^3 \times 2^3$ in Figure 2.4b. Figure 2.4c shows the maximal square block decomposition, i.e., the quadtree decomposition, of the image in 2.4b. The process of this block decomposition can



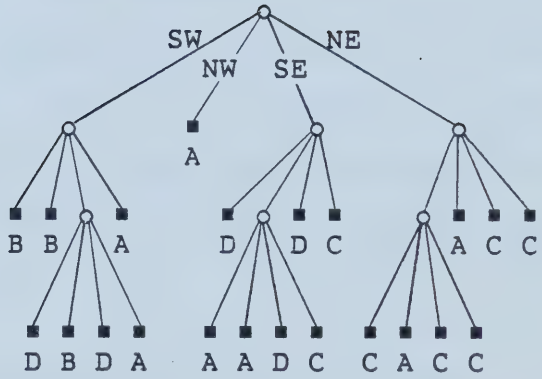
(a) A Polygon Network

A	A	A	A	A	A	C	C
A	A	A	A	A	A	C	C
A	A	A	A	A	C	C	C
A	A	A	A	C	C	C	C
B	B	A	A	A	C	C	C
B	B	A	A	A	D	C	C
B	B	B	A	D	D	D	D
B	B	D	D	D	D	D	D

(b) An Array Representation



(c) A Maximal Block Representaion



(d) A Tree Representation

Figure 2.4. A Polygon Network and Its Representations

be represented by a tree as shown in Figure 2.4d. In this tree, the root node corresponds to the entire image, each son of an internal node represents a quadrant of the image represented by that internal node, and leaf nodes correspond to homogeneous blocks for which no further subdivision is required.

As for the size of the quadtree, it has been reported [28] that, for a simple polygon of perimeter p (measured in pixel widths) and a resolution n (the number of levels in the quadtree), the number of nodes in the quadtree is of $O(p + n)$. That is, the number of quadtree nodes depends mainly on the summed length of the boundaries measured at a chosen resolution. It is therefore guaranteed that the tree grows linearly as either the resolution or boundary length is doubled. This implies that the quadtree representation is not useful for images in which the amount of aggregation is minimal like a checkerboard.

As shown in Figure 2.4d, a natural way to encode a quadtree is to use a tree structure. Then, many basic operations, particularly set operations, can be implemented as tree traversals. Nevertheless, the tree structure may not be an ideal way of encoding the quadtree as it has a considerable amount of storage overhead associated with it. First, the tree implementation of a quadtree has an overhead in terms of internal nodes. A quadtree with m leaf nodes has $(m-1)/3$ internal nodes. Another kind of storage

overhead arises from the need to store the pointers of the tree. Each node in the tree is represented as a record with five pointers, four to the records corresponding to its sons and one to the record corresponding to its father.

Naturally, there has been a lot of effort to reduce the storage requirement of the quadtree. The resultant schemes are *pointerless quadtrees*, and they can be grouped into two categories: first, those treating the image as a collection of leaf nodes, and second, those representing the image in the form of a traversal of the nodes of the quadtree.

The first group of pointerless quadtrees are referred to as *linear quadtrees* and originated from Gargantini [13]. In a linear quadtree, each leaf is encoded by a base 4 number, called a locational code or a key, that corresponds to a sequence of directional codes that locate the leaf along a path from the root of the quadtree. The result is equivalent to taking the binary representation of the x and y coordinates of a designated pixel in a block and interleaving them. This technique is known as Morton indexing [28]. A linear quadtree has an interesting property that when the keys of the leaf nodes are listed in ascending order, the resulting sequence is in a postorder (in fact, preorder or inorder as well, since internal nodes are not included) traversal of the quadtree blocks. The following is the linear quadtree encoding of the quadtree introduced in Figure 2.4:

< 0,2,B>	< 4,2,B>	< 8,3,D>	< 9,3,B>
<10,3,D>	<11,3,A>	<12,2,A>	<16,1,A>
<32,2,D>	<36,3,A>	<37,3,A>	<38,3,D>
<39,3,C>	<40,2,D>	<44,2,C>	<48,3,C>
<49,3,A>	<50,3,C>	<51,3,C>	<52,2,A>
<56,2,C>	<60,2,C>		

Each leaf is represented by a 3-tuple of <key, level, polygon name>[3]. For each leaf, the lower left corner element of the corresponding block has been chosen for the construction of the key. The south-west corner of the image is assumed to be the origin of the coordinate space.

The second type of pointerless representation is achieved by listing the nodes of the quadtree, both internal nodes and leaves, in a preorder, i.e., depth first, traversal of the tree. The resulting sequence is a string which consists of a polygon name for each leaf node and a special symbol for an internal node. This encoding technique is referred to as a *DF-expression* and is due to Kawaguchi and Endo [15]. Using a DF-expression, the quadtree in Figure 2.4d is encoded as follows:

((BB(DBDAAA(D(AADCDC((CACCACC,

where '(' represents an internal node. Compared to linear quadtrees, the DF-expression is more efficient spacewise, but it is not as context free as linear quadtrees. A binary tree variant of the DF-expression is discussed in [33].

---

[3] This representation is from [27] and is utilized to represent a polygon network in Chapter 4 of this thesis.

2.4 File Structures

The primary concern in organizing polygon network data in a file is the access efficiency. Much study has been done on access mechanisms for multi-dimensional data, but most of it was focused on point data. Spatial access to point data in a k-dimensional coordinate space is analogous to multi-key access to records with composite keys in the sense that coordinates can be used together as a key. Access to region data, however, is more problematic because exact-match queries are almost impossible on region data. In other words, no predetermined keys corresponding to access are known *a priori* to a query. For instance, it is not possible to store all the potential points that may be requested for a point-in-polygon problem. Instead, a point-in-polygon problem is solved by referring to the information in the neighborhood of the given point.

Therefore, it is desired that all the information required to solve a given problem is stored physically together. Furthermore, it should be ensured that only those parts of the file that are spatially relevant to processing a query are retrieved. For these reasons, an efficient file structure must preserve locality. That is, objects that are near to each other on the study area should also be near to each other in the file with a high probability of being stored in the same physical disk block.

In this regard, cell methods are the most attractive because cells capture locality. The cell method organizes data by cells which are produced by dividing the study area into a regular grid of intervals corresponding to units of external storage such as disk blocks. Cells are easily accessible by address computation techniques. As the basic unit of a file, a cell is expected to contain a complete description of its geometric contents so that a query (or each subproblem of an original query) can be solved with the information contained in that cell.

Then, by restricting the problem space within a cell, cell methods allow spatial queries to be answered efficiently. For example, a point-in-polygon query can now be answered by first determining the corresponding cell and then, within that cell, by either searching (in the case of raster representation) or computing (in the case of vector representation). For those queries which require the entire file to be processed, e.g., a polygon overlay problem, cell methods achieve efficiency by allowing the query to be decomposed into several subproblems within cell restrictions. In that sense, the cell method can be regarded as a divide-and-conquer technique. As for range search, cell methods are obviously the most useful because they organize data by the embedding search space.

Now, suppose that data are not uniformly distributed over the search space, i.e., in some parts of the study area

polygons are more dense than in other parts. Then, the cell method described previously (referred to as a *fixed* cell method) is not efficient because data buckets are unevenly filled, resulting in overflows and/or underflows. Therefore, it is desired that cell methods shall have the flexibility of adapting their shape automatically for non-uniformly distributed data or dynamically changing data. This can be achieved with an *adaptable* cell method in which cell boundaries are drawn at places that are fixed regardless of the contents of the file, but each boundary line is activated or deactivated depending on the data density within a cell and in its neighbourhood.

In some sense, adaptable cell methods are similar to quadtree structures discussed in the previous section. Directly importing the quadtree structure to file organization schemes, however, will cause problems because following pointers may lead to page faults. In fact, the problems related with pointers are inherent in every tree structured file scheme. Therefore, much attention has been paid to organizing files without using pointers, and some of the pointer-free file organization schemes will be explored in the following chapter.

In conclusion, the following properties should be considered in designing a file structure for polygon network data for the efficient handling of spatial queries:

(1) it should allow efficient random access to a record in

the file in order to handle point queries, e.g., point-in-polygon;

(2) it should efficiently support sequential retrieval of data to handle those queries which require the entire file to be processed in sequential order, e.g., polygon overlay;

(3) it should have an adaptable structure to handle unevenly distributed data or dynamic data; and

(4) it should have a localization feature so that a query can be decomposed into a set of subproblems.

Since the last feature is best met by a cell method, it is desirable to develop a cell method such that it has the first three properties as well.

CHAPTER 3

DYNAMIC HASHING AND CELL METHODS

This chapter is devoted to a background study for the design of a file organization scheme for structuring polygon network data. The discussion is focused on dynamic hashing, particularly extendible hashing and linear hashing, which turns out to be very useful for implementing cell methods. In addition, prior cell methods based on dynamic hashing are introduced, and their relative merits are discussed. Based on this discussion, a method that would provide a compromise between various tradeoffs is suggested.

3.1 Introduction

To be effective, a file organization scheme should allow both random and sequential access to records in the file and be able to support dynamic file maintenance. With the advent of direct access storage devices, several efficient file organization schemes have emerged ([18] provides a survey on basics). Among them, balanced tree organizations and hashing schemes have been the most successful.

A tree structure is inherently dynamic in the sense that the number of nodes in a tree varies with insertion and deletion of records. In particular, the B-tree invented by Bayer and McCreight [3] provides an efficient mechanism for structuring dynamic files. It gracefully adapts its shape

in response to insertions and deletions of records and supports retrieval of a record with $O(\log n)$ disk accesses, where n is the number of records in the file. Although this logarithmic cost of tree structured files is attractive compared with the $O(n)$ of sequential files, it is still far greater than the $O(1)$ of direct access methods. Hashing is a popular technique for organizing direct access files. If a file is static, hashing allows a record to be retrieved with one disk access in general.

Nevertheless, traditional hashing schemes have a major disadvantage in that storage allocation for the hash table is static. That is, the size of a file must be estimated in advance and storage space must be allocated for the whole file at once. Thus, a high estimate of the data volume results in a waste of space while a low estimate results in either a costly reorganization of the whole file through rehashing or the attachment of overflow buckets, which slowly degrades the $O(1)$ access time characteristic of hashing toward $O(n)$ in the worst case. Therefore, when the nature of the data is unknown or the file is dynamic, hashing has not been a good choice.

Recently, however, several *dynamic hashing* (sometimes called *virtual hashing*) schemes have been developed that overcome the difficulty in maintaining large dynamic files. Their main characteristic is that the storage space allocated to the file can be increased or decreased without

reorganizing the whole file. This is achieved by using *split* and *merge* functions to avoid the accumulation of overflow records and the underflow conditions, respectively.

Dynamic hashing schemes can be categorized into two types: those that maintain some kind of directory, the size of which varies with the file size, and those that do not use a directory. Those with a directory resolve collisions by *splitting* the overflow address. Those without a directory resolve the overflow condition by *chaining* the overflow address. A chain is *split*, however, when the global storage utilization factor exceeds a predefined upper threshold.

The most noticeable file organization schemes of the first type include *virtual hashing* by Litwin [21], *dynamic hashing* by Larson [19] and *extendible hashing* by Fagin *et al.*[11]. *Linear hashing* invented by Litwin [20] and *interpolation-based index maintenance* by Burkhard [7] are the schemes of the second type. Among these new schemes, of particular interest are extendible hashing, linear hashing and interpolation-based index maintenance that form the basis of the file organization scheme proposed in this thesis. The main features of extendible hashing and linear hashing are reviewed in the next two sections. (Interpolation-based index maintenance is not given a separate review as it is a direct generalization of linear hashing for k-dimensional data.) For more details, see [11] and [7,20] about extendible hashing and linear hashing, respectively.

3.2 Extendible Hashing

Extendible hashing has been designed to retrieve the data associated with a given key with two disk accesses, i.e., one access to retrieve the desired part of the directory and another to retrieve the data bucket that contains the given key. Extendible hashing achieves its goal by merging the concepts of a radix search tree and hashing. An intuitive description of how these two file schemes have been merged into extendible hashing is now presented.

Radix search trees, which are naturally extendible, can be modified to yield optimum speed as follows. Suppose T is a radix search tree with radix r and depth d. Then, retrieval of a key contained in a leaf at level d requires d probes. Now, suppose the top j levels of the radix search tree are flattened out in an array of r^j pointers. Then, the access to the tree can be speeded up by achieving a fanout of r^j at the root instead of the original fanout of r. Indeed, every leaf in T can be accessed in a single probe by forcing T to be a degenerate tree, i.e., by choosing j as large as d.

Next, as for hashing schemes, in order to prevent their performance from deteriorating in a dynamic environment, a mechanism which ensures that the hash table remains balanced under insertions and deletions of records is highly desirable. In extendible hashing, this is achieved by introducing the concept of a degenerate radix search tree to the

hash table as described in the following.

A hash function h maps the key space **K** onto an address space **A**. **A** is split into m blocks by a partition π, defined by $m+1$ boundaries, $\alpha_0, \alpha_1, \alpha_2, \ldots, \alpha_m$. A data bucket is allocated to each address block, and a hash table of size m implements the correspondence between data buckets and blocks. In order to make the hash table extendible while the hash function h is kept unchanged, the partition must have a variable number of variable sized blocks. It starts with a small number of blocks, e.g., $m = 1$. If a data bucket overflows, the partition is changed by introducing new boundaries so that the dense region may be covered by several data buckets.

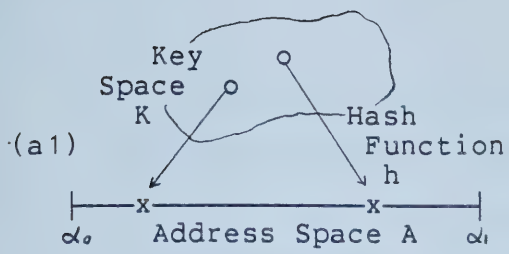
Among various partition techniques, the *buddy system partition*[4] is recommended for its simplicity. Then, when a data bucket overflows, the corresponding block in the address space is halved, a new data bucket is added, and only those keys in the halved block are reorganized. If a bucket underflows as a result of deletions and its buddy is underfilled as well, then the two buckets can be coalesced into one, decreasing the number of blocks in the partition by one. Therefore, in regions where keys cluster the partition is finer than in regions where keys are sparse.

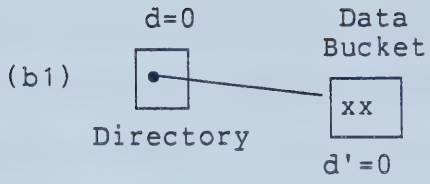
---

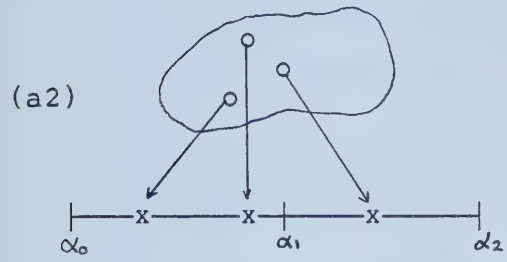
[4] Let $A = \{0,1,\ldots,2^{n-1}\}$ for some n; then the boundaries of a *buddy system partition* are $0 = a_0 < a_1 < a_2 < \ldots < a_m = 2^{n-1}$, where all intervals $[a_{i-1},a_i)$ can be obtained by repeated halving of A.

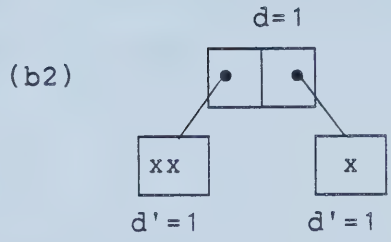
The buddy system partition of the hash address space results in the same effect as a radix search tree with radix $r = 2$ over the alphabet $\{0,1\}$. Thus, the buddy system partition can be implemented in a hash table using a directory with 2^d entries, where d is the depth of the buddy system partition, or equivalently, the depth of the radix search tree. Then, the directory allows the d leftmost bits of a key to be taken as an index to the directory. If the depth of the partition increases, the directory doubles in size.

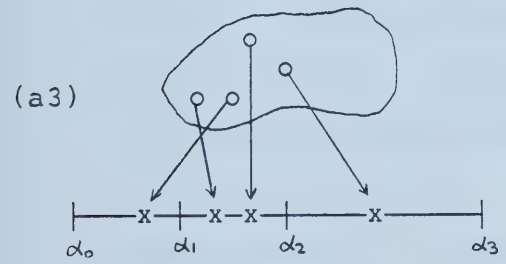
Figure 3.1 illustrates how the buddy system partition works. Assume that b, the capacity of a data bucket, is 2. Initially, the entire address space is partitioned into one block as shown in Figure 3.1a1. As more keys arrive, the buddy system partition introduces new boundaries by recursively halving the block in which a collision occurs. This process is shown in Figures 3.1a2, a3, a4, and a5, consecutively. Each of the figures in Figure 3.1b shows how the directory dynamically accommodates itself to the new partition of the address space. In Figure 3.1b, the depth of the directory is denoted by d, and the depth (i.e., level) of a data bucket is denoted by d'. Note that the keys contained in a data bucket with a depth d' would have the same bit pattern in their leftmost d' bit places, and the indexes of the directory entries which point to that bucket also have the same bit pattern in their leftmost d' bit places.

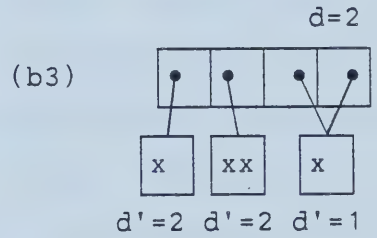


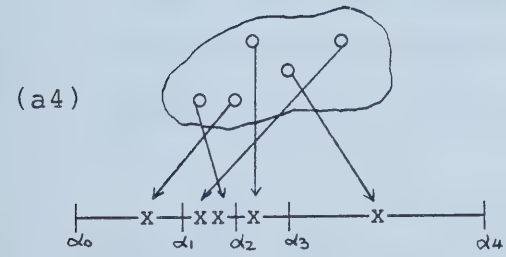


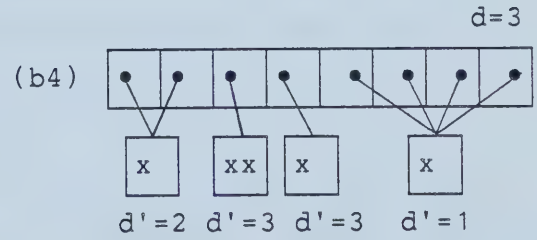


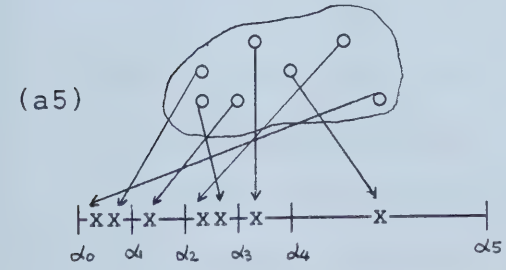












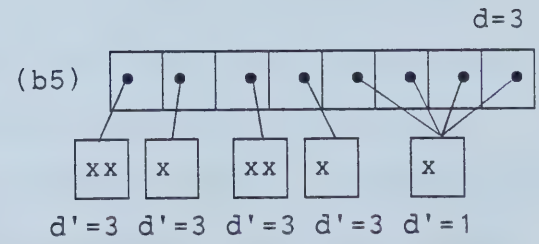


Figure 3.1. Extendible Hashing
(a) Buddy system partition of address space
(b) Directory and data bucket correspondence

Although the directory may become large and have to be kept in external storage, it is normally accessible with one disk I/O operation without using a special algorithm since the bit pattern of a key is used as an index to the directory. Therefore, each record in a file can be retrieved in two disk accesses with extendible hashing.

3.3 Linear Hashing

Linear hashing is particularly remarkable because it maintains good performance without using an explicit directory. The major advantages of linear hashing are: (1) the file size grows and shrinks gracefully, (2) there is no directory to be kept, and (3) the utilization of the allocated storage is controlled. It has been reported [20] that using linear hashing may lead to an average number of accesses per successful search not greater than 1.03, while the storage utilization ratio stays close to 60%.[5]

For an insight into the design principles of linear hashing, consider the following situation. Assume a set of records is being inserted into N buckets, indexed from 0 to N-1, using some hash function, and that a collision occurs at the insertion of a record with a key k. Furthermore, assume a bucket split is performed to avoid the accumulation of overflow records. A natural approach would be to split

---

[5] Note that the experiment was done with ordinary identifier based records. Thus, the results may not hold for geometric data.

the bucket in which the collision occurs. However, this approach calls for some sort of directory since buckets will be chosen randomly for a split.

Next, consider the following approach. Suppose that splits are performed on buckets in a predefined order, e.g., in the order of 0,1,..., N-1, regardless of which bucket a collision occurs in. That is, first, bucket 0 is split into buckets 0 and N, next, bucket 1 is split into buckets 1 and N+1, and so on, until bucket N-1 is split into buckets N-1 and 2N-1. After each of the N buckets is split, bucket 0 is chosen again for a split, this time, into buckets 0 and 2N, and so on. Since the index of the bucket to be split is predefined while the index of the bucket with a collision is random, a bucket with a collision is split not necessarily when the collision occurs but with some *delay*. In the meantime, overflow buckets are attached to resolve the collision. The result is that the file becomes progressively larger and no directory is required. Linear hashing is based on this approach. The rest of this section describes linear hashing more exactly.

Let $\mathbf{K}$ be the key space. A hash function h assigns to each record with a key k in $\mathbf{K}$ a chain $h(k)$ in which the record should be stored. A *chain* is a list of buckets, the first of which is a *primary* bucket and the rest of which are *overflow* buckets. Next, let h_0: $\mathbf{K} \rightarrow \{0,1,\ldots,N-1\}$ be the function that are used to initially load the records of the

file into N chains. Then, a sequence of functions, h_0, h_1, h_2, ..., are *split functions* for h_0, provided that, for all $i \geq 0$, both of the following requirements are satisfied:

[range condition] h_i: K $\rightarrow \{0,1,\ldots,2^iN-1\}$ and

[split condition] for each k in K, either

$$h_{i+1}(k) = h_i(k) \text{ or}$$

$$h_{i+1}(k) = h_i(k) + 2^iN.$$

The range condition specifies that the function h_i distributes the keys into 2^iN chains, and the split condition implies that a chain with an index $h_i(k)$ is split into chains with indexes $h_i(k)$ and $h_i(k) + 2^iN$ and that these indexes can be obtained by the next higher function $h_{i+1}(k)$.

In addition, the following three variables are defined.

(1) *File level*, j, is the greatest index of split functions that have been activated. Initially, when the file is loaded, j is set to 0 and the active split function is h_0. At file level $j > 0$, active functions are h_{j-1} and h_j.

(2) *Next chain to split*, p, is a pointer to the chain which will split next. p is initially 0 when the file is created but is increased by one up to $N-1$ as the first N splits are performed. During the second cycle of splits, the value of p changes from 0 to $2N-1$; during the third cycle, from 0 to 2^2N-1, and so on.

(3) *Load factor*, w, is the ratio of the number of records in the file to the number of slots available in the current set of chains. The load factor is limited by user

specified upper and lower thresholds.

Now, suppose that a record with a key k is to be located. The index of the chain, i.e., m, that contains the key k is easily computed with the following algorithm:

Step 1. if $j = 0$, #is file level 0?#
then $m \leftarrow h_0(k)$ and exit #trivial case#
Step 2. if $p = 0$, #every chain split?#
then $m \leftarrow h_j(k)$ and exit #only one function used#
Step 3. $m \leftarrow h_{j-1}(k)$ #determine chain#
Step 4. if $m < p$, #is chain split?#
then $m \leftarrow h_j(k)$ #use higher function#

The insertion of a record with a given key is also simple. First, the access algorithm determines which chain should contain the specified key. If the key is not present, the record with the given key is inserted and the load factor is updated accordingly. Next, if the load factor exceeds the upper limit, a chain split is performed. The following example illustrates how the insertion procedure works.

Suppose that a set of keys,

17,5,23,12,11,6,19,24,37,1,7,29,4,30,18,25,13,

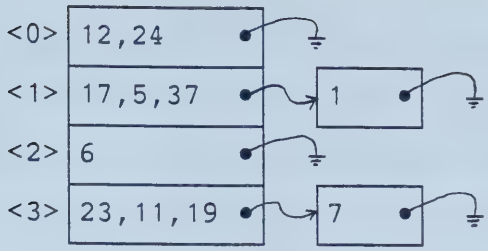
are to be inserted using the split function,

$$h_i = k \bmod 2^i N,$$

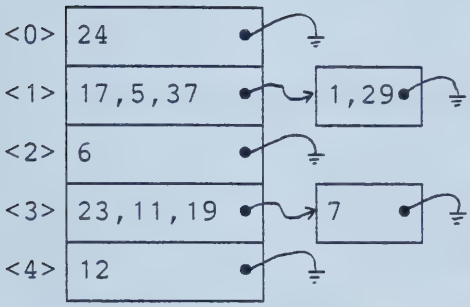
where $N = 4$. Assume that the primary bucket size $b = 3$, the overflow bucket size $b' = 2$, and the lower and upper limits on the load factor are 0.40 and 0.70, respectively. At the

initial construction of the file, four chains are created, and both the file level j and the next chain to split p are set to 0. Figure 3.2a shows the configuration of the file after the keys 17, 5, 23, 12, 11, 6, 19, 24, 37, 1, and 7 have been inserted with $h_0 = k \bmod 4$. At this stage, the load factor $w = 11/16 = 0.69$. At the insertion of the next record with the key "29", however, the load factor w becomes $12/16 = 0.75$, exceeding the upper limit. Therefore, chain 0 is split into chains 0 and 4 using $h_1 = k \bmod 8$ as shown in Figure 3.2b. Both the file level j and the next chain to split p become 1. Next, after the insertion of the keys "4" and "30", chain 1 is split into chains 1 and 5, moving the overflow records "1" and "29" into primary buckets. The result is shown in Figure 3.2c. Similarly, the insertion of "18" and the insertion of "25" and "13" cause chains 2 and 3 to be split, respectively, as shown in Figure 3.2d and 3.2e. Now, one cycle of bucket splitting has been completed, and the file configuration in Figure 3.2e looks as if it were loaded initially with h_1. Notice that the next chain to split is 0 again and that the next higher split function to be activated is $h_2 = k \bmod 16$, should the need arise.

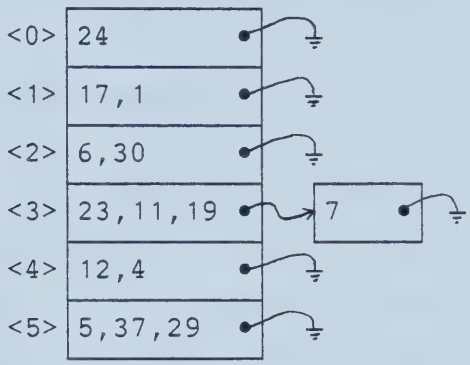
The deletion operation is similar to insertion. First, the access algorithm is utilized to determine which chain contains the key to be deleted. If the key is found in the chain, it is removed from the file and the load factor is updated. Then, if the load factor falls below the lower limit, merging of two chains is triggered.



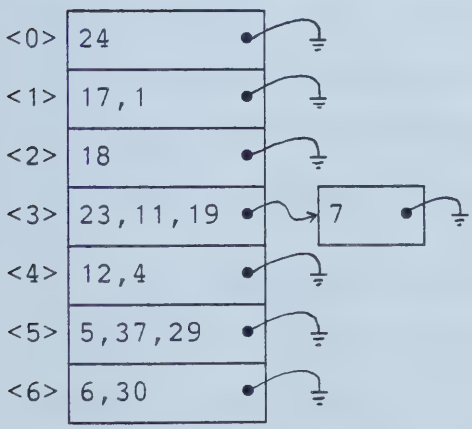
(a) After inserting the first eleven keys:
j=p=0; w=11/16.



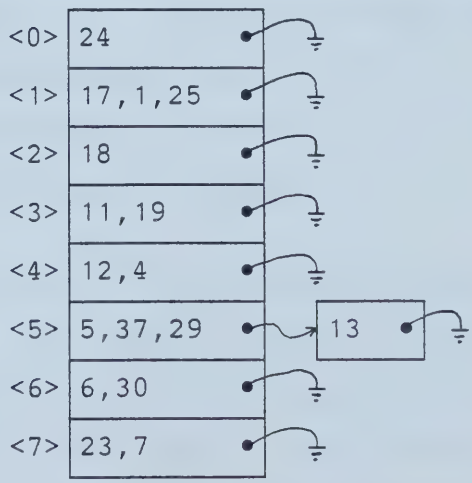
(b) After inserting "29":
j=p=1; w=12/19.



(c) After inserting "4","30":
j=1; p=2; w=14/20.



(d) After inserting "18":
j=1; p=3; w=15/23.



(e) After inserting "25","13":
j=1; p=0; w=17/26.

Figure 3.2. Linear Hashing

As illustrated, with linear hashing the allocated storage increases linearly, i.e., one chain at a time, maintaining a good overall storage utilization. Although overflow keys are moved to primary buckets with some delay, the rate of overflow keys usually remains small if the bucket capacity $b >> 1$ [20]. Thus, it is reasonable to expect that a key is normally found in one disk access.

3.4 Cell Methods Based on Dynamic Hashing

Both extendible hashing and linear hashing lend themselves to an adaptable cell method of structuring k-dimensional data. In fact, most of the successful cell methods that have been implemented are multi-dimensional variations of either extendible hashing or linear hashing. The *EXCELL* method of Tamminen [31,32,34] and the *grid file* of Nievergelt, Hinterberger and Sevcik [22] are adaptations of extendible hashing to k-dimensional data, and the *interpolation-based index maintenance* of Burkhard [7] is a k-dimensional generalization of linear hashing.

The *EXCELL* method and the *grid file* achieve an upper bound of two disk accesses in retrieving a record in the same way as extendible hashing does. Their approach is to use a geometric directory which maintains a dynamic correspondence between the cells of the search space and data buckets.

The major difference between the EXCELL method and the grid file results from their different way of partitioning the search space. The EXCELL method employs the buddy system partition in which cell division lines are all equidistant. In contrast, the grid file does not require cell division lines to be equidistant. Hence, a directory of the grid file consists of two parts: a dynamic k-dimensional array containing pointers to data buckets and a set of k 1-dimensional arrays termed *linear scales*. The linear scales define the partition of each axis and allow the pointers to be located in the directory by computing the cell addresses from the given coordinates.

Another difference between the EXCELL method and the grid file is found in their splitting strategy. With the grid file, the grid refinement operation divides a single interval for the particular axis into two, resulting in the insertion of a $(k-1)$-dimensional cross section. The grid refinement operation with the EXCELL method, on the other hand, divides *every* interval for the particular axis into two, which results in a doubling of the directory size. Therefore, with the use of the grid file, the directory does not grow so fast. The use of the EXCELL method, however, simplifies grid refinement operations at the expense of larger directories in general.

Next, with the EXCELL method, the split occurs perpendicular to the different axes in a *cyclic* fashion; for

example, in a 2-dimensional case, the first split occurs perpendicular to the *x*-axis, then to the *y*-axis, then to the *x*-axis, and so on. On the other hand, with the grid file, one is given a choice at the grid partition refinement with respect to the axis as well as the location of the splitting point. Due to this flexibility, the grid file is not so sensitive to the pattern of data distribution as the EXCELL method.

Recently, Tamminen [34] suggested a hierarchical EXCELL method as a solution to the problem of a large directory requirement. In a hierarchical EXCELL scheme, each cell may correspond either to a data bucket or a directory of another hierarchical EXCELL file. At the maximum depth of the hierarchy, overflow is allowed if a directory would otherwise contain more than a predefined number of elements per data bucket. It should be noted that this approach does not guarantee an upperbound of two disk accesses for retrieving a record.

Interpolation-based index maintenance (hereafter, *interpolation hashing*) is a multi-dimensional variation of linear hashing. It has been designed to handle range queries efficiently while preserving the performance of linear hashing in terms of disk accesses required for insertion, deletion and retrieval of a record. Similar to the EXCELL method and the grid file, interpolation hashing handles range search efficiently by associating each chain,

i.e., each data bucket, with a cell in the search space.

The association between chains and cells of the search space has been made available to interpolation hashing through modification of the split conditions of linear hashing. When a chain is split into two, the split conditions of interpolation hashing distributes the keys into two chains such that the smaller keys remain in the same chain while the larger ones move to the new chain. In other words, the split conditions of interpolation hashing examine the leftmost bits of the key to determine which chain should contain the key, in the same way as the EXCELL method.

Another similarity of interpolation hashing to the EXCELL method is found in the way it implements hashing. Both methods produce keys for k-dimensional data by shuffling their k coordinates, i.e., by taking the binary representation of the coordinates for each of the k axes and interleaving them. The shuffling of coordinates, together with the modified split conditions, produces for interpolation hashing a grid partition of the search space at intervals determined by a radix. Consequently, as in the EXCELL method, cell division lines with interpolation hashing are always equidistant, and the axis for a split is chosen in a cyclic fashion. Hence, the cells of the search space have the same shape as those of the EXCELL method; namely, the cells are either square or rectangular with their length

twice as long as their width.

The major difference between interpolation hashing and the EXCELL method is that the correspondence between cells and chains is given by formulas rather than by a directory. Furthermore, with interpolation hashing the relationship between chains and cells of the study area is a one-to-one mapping in contrast to the one-to-many mapping between buckets and cells with the EXCELL method. It should be also noted that with interpolation hashing there exist at most two different sizes of cells at any moment. Besides, with interpolation hashing the number of cells of the search space grows by one while with the EXCELL method the number of cells grows by doubling.

Figure 3.3 illustrates the differences among various partitions with an example of a 2-dimensional search space. Notice that, with the EXCELL method and interpolation hashing, a cell division line is always at the midpoint of an interval and alternately perpendicular to the x and y axes. However, the grid file does not follow this convention. As a result, the cells with the grid file are not compatible with the blocks of a quadtree. On the other hand, the shape of cells with the EXCELL method and interpolation hashing is compatible with that of quadtree blocks. It should be noted, however, that the use of interpolation hashing may require splitting and/or merging of quadtree leaves since there is no sharing of a data bucket among

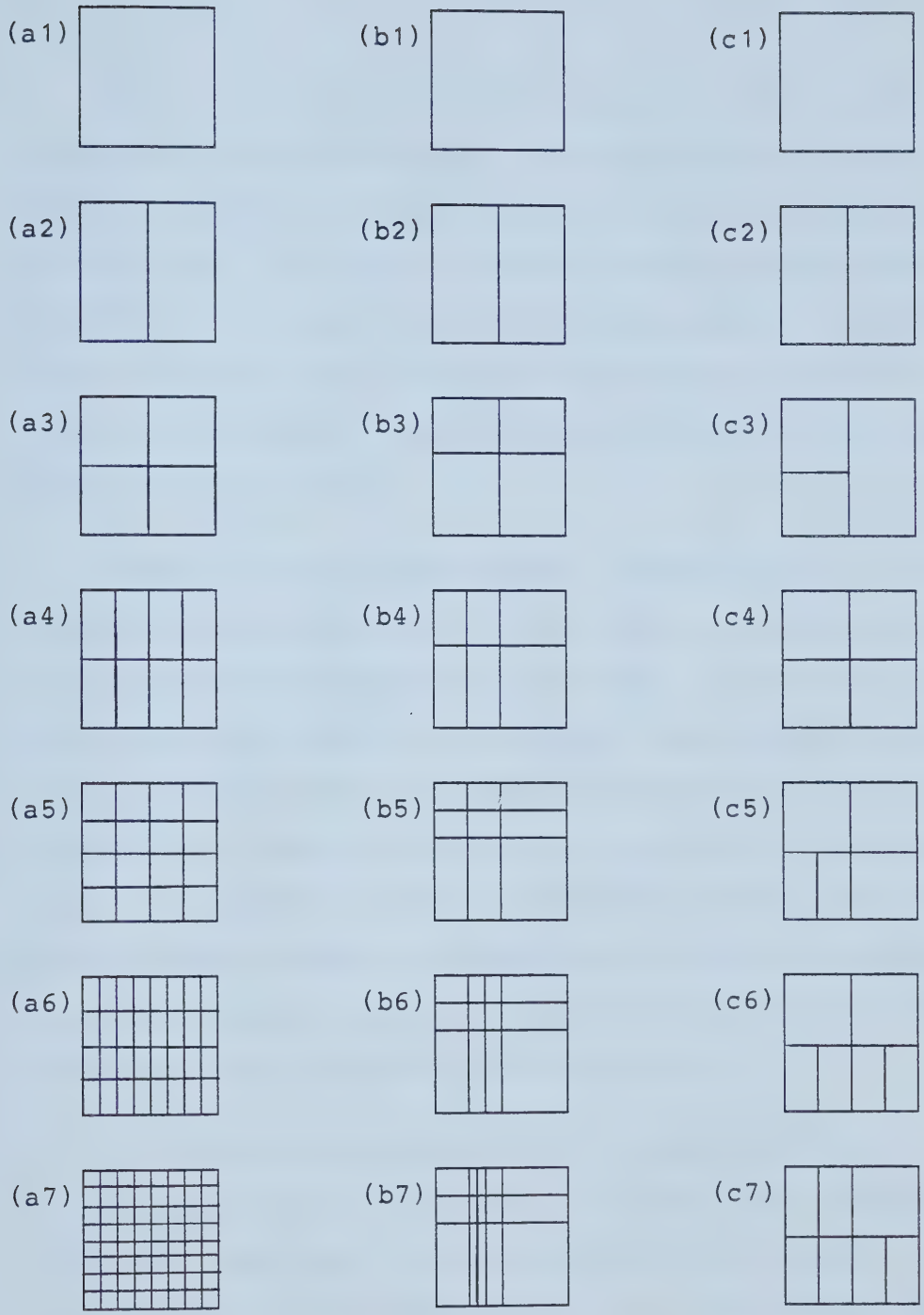


Figure 3.3. Comparison of Search Space Partitions
(a) EXCELL method (b) Grid file method
(c) Interpolation-based index maintenance

several cells.

Each of the cell methods discussed so far has its advantages and disadvantages. Methods based on linear hashing provide for linear growth of the file without using a directory. In contrast, methods based on extendible hashing have overhead in terms of a directory. Particularly, with the EXCELL method, the directory may become large and unwieldly if the data are not evenly distributed over the search space.

Linear hashing based methods, however, usually have overflow buckets attached to their chains and may result in a file with unbalanced bucket occupancy. Especially, a non-uniform distribution of data may force linear hashing based methods to keep many overflow buckets as well as many underfilled primary buckets at the same time. On the other hand, with methods based on extendible hashing, the occupancy of data buckets is significantly more uniform because extendible hashing leads to a partition of the search space into blocks of a variable number of cells.

Each method behaves well for the data that are uniformly distributed over the study area. The worst case occurs when the data are non-uniformly distributed. In order to improve the performance in the worst case, it is desirable to develop a method that provides a compromise in the tradeoff between extendible hashing based methods and linear hashing based methods, namely, the tradeoff between

the overhead of maintaining a large directory and the advantages of balancing bucket occupancy.

If it can be assumed that the data density of a particular region is more or less the same as that of its neighbours, a file structure that combines the principles of extendible hashing and linear hashing may be useful. That is, a new cell method may be developed by modifying linear hashing so that it has a directory and resolves collisions by splitting the address space instead of chaining until the directory reaches a predefined maximum depth. Typically, the maximum depth of a directory will be defined such that the directory may reside in main memory. Equivalently, a new method may be developed by modifying extendible hashing so that when the depth of the directory exceeds the predefined value, chaining of the overflow address is used instead of indefinitely expanding the directory. This new cell method will reduce the effect of a single local variation of data density on the file structure, achieve more uniform bucket occupancy than when linear hashing alone is applied, and retrieve each record in a file with one disk access in general. From this point of view, a new cell method which merges the concepts of extendible hashing and linear hashing is proposed in the next chapter. The resulting file scheme incorporates the features of both the EXCELL method and interpolation hashing.

CHAPTER 4

A PROPOSED FILE ORGANIZATION SCHEME

This chapter presents a file organization scheme for representing a polygon network of geoemtric data by a quadtree. The scheme is an adaptive cell method which has been developed based on extendible hashing and interpolation hashing. An adaptive cell method is first defined. Then, the main features of the method, i.e., the mapping between data buckets and the regions in the study area and the dynamic nature of its file maintenance, are discussed. Finally, a description of how the proposed scheme supports various spatial queries is presented.

4.1 Definitions and Notations

Let the study area,

$$U = [0, 2^n)^2,$$

be an image of $2^n \times 2^n$ unit square pixels that intersects a polygon network, and let each of the pixels have a polygon name (hereafter called *color*) associated with it. Furthermore, let the polygon network on **U** be represented by a region quadtree.

As there does not exist a natural total order among the blocks of a quadtree, or in multi-dimensional data in general, a function which maps k-dimensional data into a 1-dimensional space that admits a total order is required.

In addition, for efficient spatial access to data, the mapping should be order preserving. Thus, the following properties are required of the mapping function: (1) it should yield some arbitrary but consistent total order among the data, and (2) it should retain the spatial relationships that are inherent in the data. In this regard, the following hash function is introduced.

Definition 1. Let $(x,y) \epsilon \mathbf{U} = [0,2^n)^2$ be the x and y coordinates of the lower-left corner of a block of a quadtree defined on **U** and have the following binary representation:

$$x = \Sigma\, a_i 2^i, \text{ for } 0 \le i \le n-1, \text{ and}$$

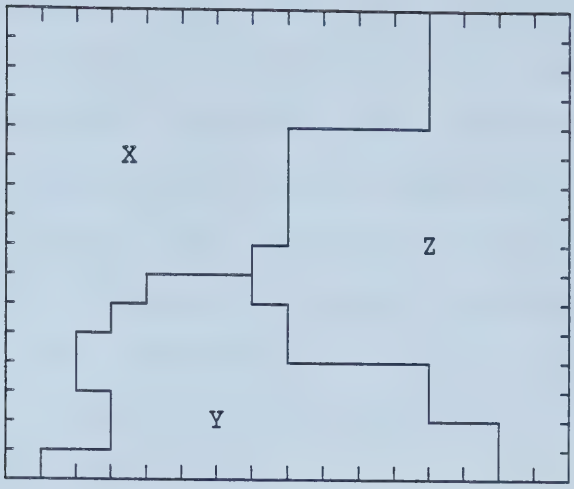
$$y = \Sigma\, b_i 2^i, \text{ for } 0 \le i \le n-1,$$

where $a_i, b_i \epsilon \{0,1\}$. Then, an *order preserving hash function*, S, that maps (x,y) onto the key space of $[0,4^n)$ is defined by:

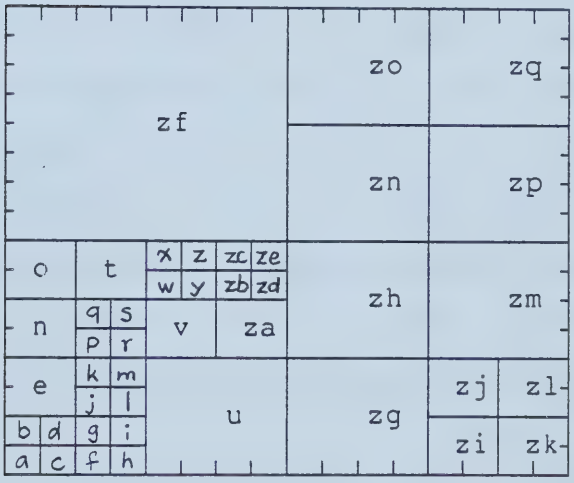
$$S(x,y) = \Sigma(a_i 2^{2i+1} + b_i 2^{2i}), \text{ for } 0 \le i \le n-1. \quad \blacksquare$$

Notice the key produced for each of the blocks in this manner is a fixed length ($2n$ bits long) integer and it is essentially the same as the locational code of the block in linear quadtree encoding.

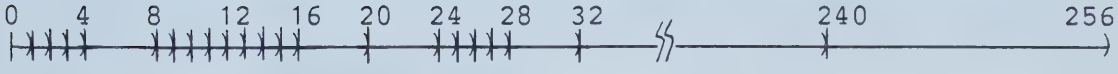
An example is given in Figure 4.1 to illustrate how a polygon network in a 2-dimensional space is transformed into a set of intervals in a 1-dimensional space using the function S. Figure 4.1a is an image of size $[0,2^4)^2$ representing a polygon network whose quadtree block decomposition



(a) Image of a polygon network



(b) Block decomposition of the image



abcd e fghijklm n o pqrs t u zp zq

(c) Intervals corresponding to the blocks

Figure 4.1. 1-Dimensional Transformation of Polygon Network Data

is shown in Figure 4.1b. Each block in Figure 4.1b has been named using the lower case alphabet for convenience. Notice that the lower-left corner of the image is chosen for the origin. By applying S to each of the blocks, they can be arranged in a total order as shown in Figure 4.1c. Indeed, each block of the quadtree in the 2-dimensional space of $[0,2^4)^2$ has been transformed into an interval in the 1-dimensional space of $[0,4^4)$, e.g., *a*:[0,1), *b*:[1,2), *e*:[4,8), *zq*:[240,256), etc.

One way of identifying these 1-dimensional intervals is to associate each of them with a pair of numbers, i.e., the coordinate of the leftmost end point and the length of the interval such as: *a*:$(0,4^0)$, *b*:$(1,4^0)$, *e*:$(4,4^1)$, and *zq*:$(240,4^2)$. The length of an interval is the same as the number of pixels that intersect the corresponding block on the image. Adopting this notation results in the following definition of a file that represents a polygon network by a quadtree.

Definition 2. A file **F** representing a polygon network for the study area **U** by a quadtree is the set,

$$\mathbf{F} = \{(K(L),S(L),C(L)): L \epsilon R\},$$

where R is the set of the leaf nodes (blocks) of a region quadtree on **U**,

K(L) is the key of L produced by *S*,

S(L) is the size, or alternatively the level, of L, and

C(L) is the color of the pixels intersecting L. ■

In order to structure the file **F** using an adaptive cell method, the study area is partitioned into a set of blocks and/or subblocks which are defined in the following.

Definition 3. A *block* of *depth d*, $0 \le d \le maxd \le 2n$, where *maxd* is the predefined maximum depth of a block partition, is a rectangular region in the study area with a standard shape and a standard location that are the same as those of a region produced by recursively halving the study area *d* times with lines alternately perpendicular to the *x* and *y* axes. A block with its depth equal to *maxd* is called a *minimal block*. ■

Throughout this thesis, *maxd* will be used to denote the predefined maximum depth of a block partition. A block is in fact a collection of one or more adjacent cells which are obtained in the following manner.

Definition 4. At depth *d* of a cell partition, where *d* is the same as the maximum of the depths of existing blocks, the study area is partitioned into 2^d *cells* that are obtained by overlaying on the study area a regular grid of *x*-spacing 2^{n-dx} and *y*-spacing 2^{n-dy}, where $dx = \lceil d/2 \rceil$ and $dy = \lfloor d/2 \rfloor$. ■

A minimal block may be further partitioned into a set of subblocks.

Definition 5. A *subblock* of *depth d*, where $maxd < d \le 2n$, is a rectangular region in the study area with a shape and a

location that are the same as those of a region produced by recursively halving the study area d times with lines alternately perpendicular to the x and y axes. Within each *minimal block*, there exist at most two different depths of subblocks, i.e., d' and d'' such that $d'' = d'+1$. ■

The following definition is useful for defining an adaptive cell method.

Definition 6. A *fixed data bucket* is a bucket which contains no more than a predefined number of records, b, and an *expandable data bucket* is a bucket which may contain more than b records by attaching one or more overflow fields to it. ■

An adaptive cell method is now defined that organizes the leaf nodes of a region quadtree into a file.

Definition 7. An *adaptive cell method* of organizing a file **F** is an abstract data type:

(1) which guarantees that, for every cell G1 and G2 and for every record L1 ϵ **F** $\cap$ G1 and L2 ϵ **F** $\cap$ G2,

key(L1) < key(L2) if index(G1) < index(G2),

(2) which guarantees that, for every subblock G1' and G2' of a minimal block G and for every record L1' ϵ **F** $\cap$ G1' and L2' ϵ **F** $\cap$ G2',

key(L1') < key(L2') if index(G1') < index(G2'),

and

(3) which asserts that every block of depth *d* has exactly one fixed data bucket associated with it if *d* < *maxd*; otherwise (the case of a minimal block), it has associated with it either a single fixed bucket or two or more expandable buckets that are contiguously located in physical memory such that

a) each expandable bucket is associated with exactly one subblock of the minimal block and

b) the overall load factor, i.e., the ratio of the number of existing records to the number of slots available, among these expandable buckets is within some predefined range. ■

4.2 Mapping between Regions in the Study Area and Data Buckets

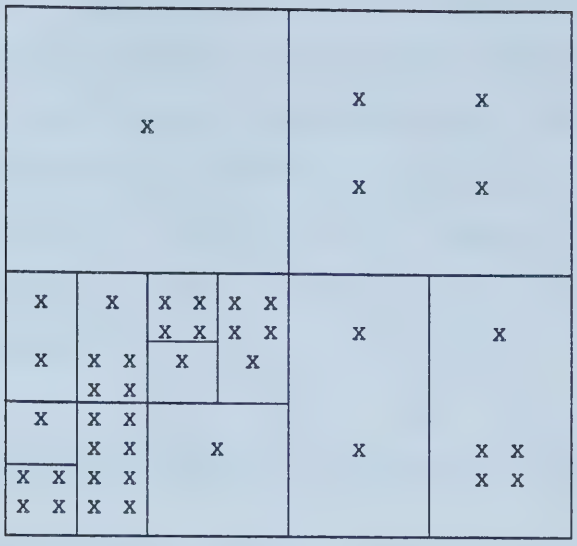
Ordinarily the set of records in **F** is distributed over a number of data buckets, and each data bucket has associated with it a block or a subblock in the study area. Therefore, a mechanism is required to locate a data bucket corresponding to each block or subblock. The mapping between blocks and data buckets is achieved by a directory while the mapping between subblocks and data buckets is achieved by computing the corresponding bucket number for each subblock. After the characteristics of data buckets are discussed briefly, these two types of mappings are described.

4.2.1 Characteristics of Data Buckets

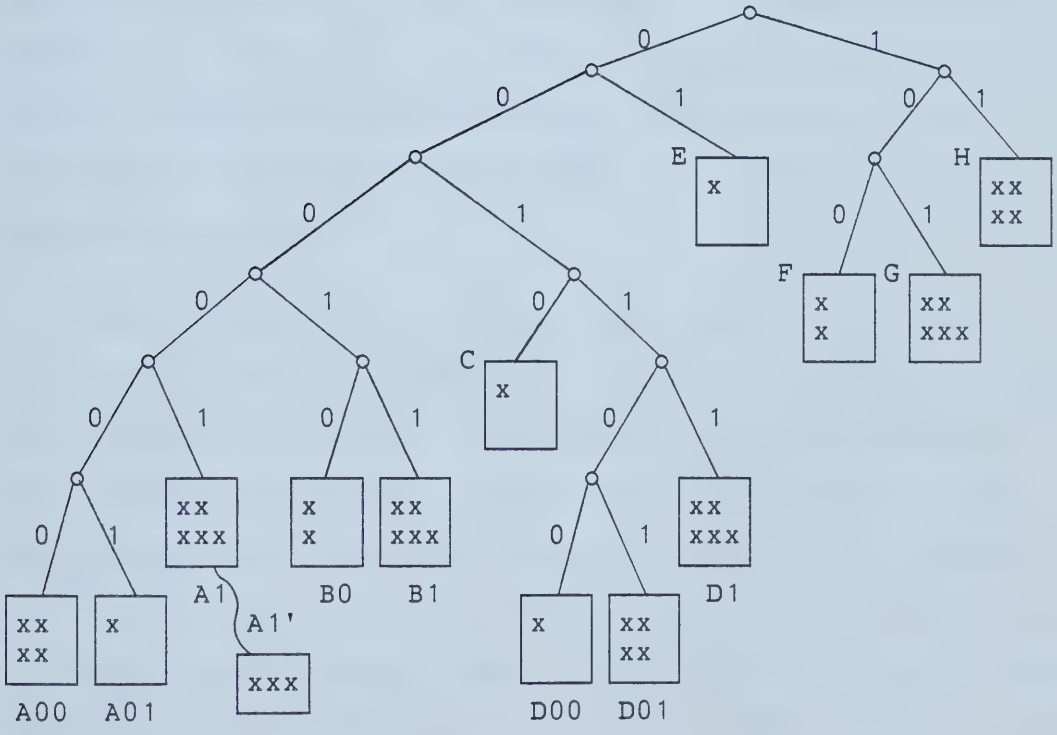
Every fixed data bucket is associated with a block in the study area, whereas every expandable data bucket is associated with a subregion of a minimal block, i.e., a subblock. Consequently, each data bucket has associated with it a depth which is equal to the depth of the block or the subblock it corresponds to.

The whole concept of a data space partition is found in Definition 7. Figure 4.2 illustrates a partition of the data space using the image of the polygon network introduced in Figure 4.1. Let the predefined maximum depth of a block partition *maxd* = 4 (*maxd* is normally small so that the directory may be stored in main memory), the capacity of a data bucket b = 5, the capacity of an overflow field b' = 3, and the lower and upper limits of the load factor be 0.40 and 0.75, respectively. Then, for the given image, the proposed scheme produces a partition as shown in Figure 4.2a ('x' denotes the presence of a record describing a quadtree block). Figure 4.2b shows the partition as a tree.

Notice that thirteen buckets and one overflow field are used to store the data. Buckets *C*, *E*, *F*, *G* and *H* are fixed buckets, and buckets *A00*, *A01*, *A1*, *B0*, *B1*, *D00*, *D01*, and *D1* are expandable buckets. Note that all the records contained in a data bucket of depth d have the same bit pattern in their first d bit places. Thus, buckets *F* and *G*, whose depths are 3, consist of the records whose keys start with



(a) Planar representation



(b) Tree representation

Figure 4.2. Partition of Data Space

"100" and "101", respectively, while buckets *A00* and *A01*, whose depths are 6, contain all the records whose keys start with "000000" and "000001", respectively. Also, notice that buckets *A00*, *A01* and *A1* contain 13 records in total while their capacity is 18. Thus, the load factor of these three expandable buckets is 13/18 = 0.72, which is within the predefined range.

4.2.2 Correspondence between Blocks and Data Buckets

The mapping between blocks and data buckets is achieved by a directory. A directory is a set of elements, each of which corresponds to a cell of size 2^{2n-d}, where d is the depth of a cell partition. Thus, a directory has associated with it a depth which is the same as the depth of a cell partition. Consequently, the depth of a directory is not greater than *maxd*.

Each element of a directory has a pointer to a data bucket or a set of buckets which contains records describing the quadtree leaf nodes that intersect the corresponding cell in the study area. At depth d of a directory, there are altogether 2^d pointers, indexed from 0 to 2^d-1, which are not necessarily unique. The pointers of a directory are indexed in such a manner that a data bucket or a set of data buckets pointed to by a pointer with an index i contains all the records whose keys are prefixed with bits that are identical to the binary representation of i. More explicitly, a

data bucket or a set of data buckets pointed to by pointer 0 contains all the keys that start with d consecutive "0" bits, a data bucket pointed to by pointer 1 contains all the keys that start with $d - 1$ consecutive "0" bits followed by a "1" bit, and so on, until finally a data bucket pointed to by the last pointer, i.e., pointer 2^d-1, contains all the keys that start with d consecutive "1" bits. Thus, the pointer i is guaranteed to find all the keys whose first d bits agree with the binary representation of i.

The above indexing scheme is in fact equivalent to a Morton sequence and naturally satisfies the first requirement of Definition 7. Figure 4.3 illustrates the correspondence between the cells in the study area and directory elements (indexes are shown both in decimal and binary). Note that when the depth of a directory is odd, each pair of buddies at the deepest level are numbered consecutively from

[2] 010	[3] 011	[6] 110	[7] 111
[0] 000	[1] 001	[4] 100	[5] 101

(a) Depth of directory 3

[5] 0101	[7] 0111	[13] 1101	[15] 1111
[4] 0100	[6] 0110	[12] 1100	[14] 1110
[1] 0001	[3] 0011	[9] 1001	[11] 1011
[0] 0000	[2] 0010	[8] 1000	[10] 1010

(b) Depth of directory 4

Figure 4.3. Correspondence between Directory Elements and Cells in the Study Area

left to right, e.g., the pair 0 and 1 and the pair 2 and 3 in Figure 4.3a.

The mapping between directory elements and data buckets, or sets of data buckets, is many-to-one. Figure 4.4 shows the directory configuration corresponding to the partition shown in Figure 4.2. The directory has 2^4 pointers because the largest of the depths of existing blocks is 4 which is the same as *maxd*. Bucket *C* is pointed to by pointer 2 and contains all the records whose first four bits are equal to "0010". In contrast, bucket *E* is pointed to by four pointers, i.e., 4, 5, 6 and 7. That is, bucket *E* contains all the records that start with "0100", but it contains all the records that start with "0101", "0110", and "0111", as well. In other words, it contains all the records whose first two bits are "01". Thus, bucket *E* is depth 2 while bucket *C* is depth 4.

Note that pointer 0 points to a set of buckets, i.e., buckets *A00, A01 and A1*, which contains all the records that start with "0000". Due to the density of keys in this area, the corresponding minimal block has been further partitioned into three subblocks and three expandable data buckets have been employed. The correspondence between subblocks and expandable data buckets, however, is not shown in the directory. The following section describes a method providing the mapping of this kind.

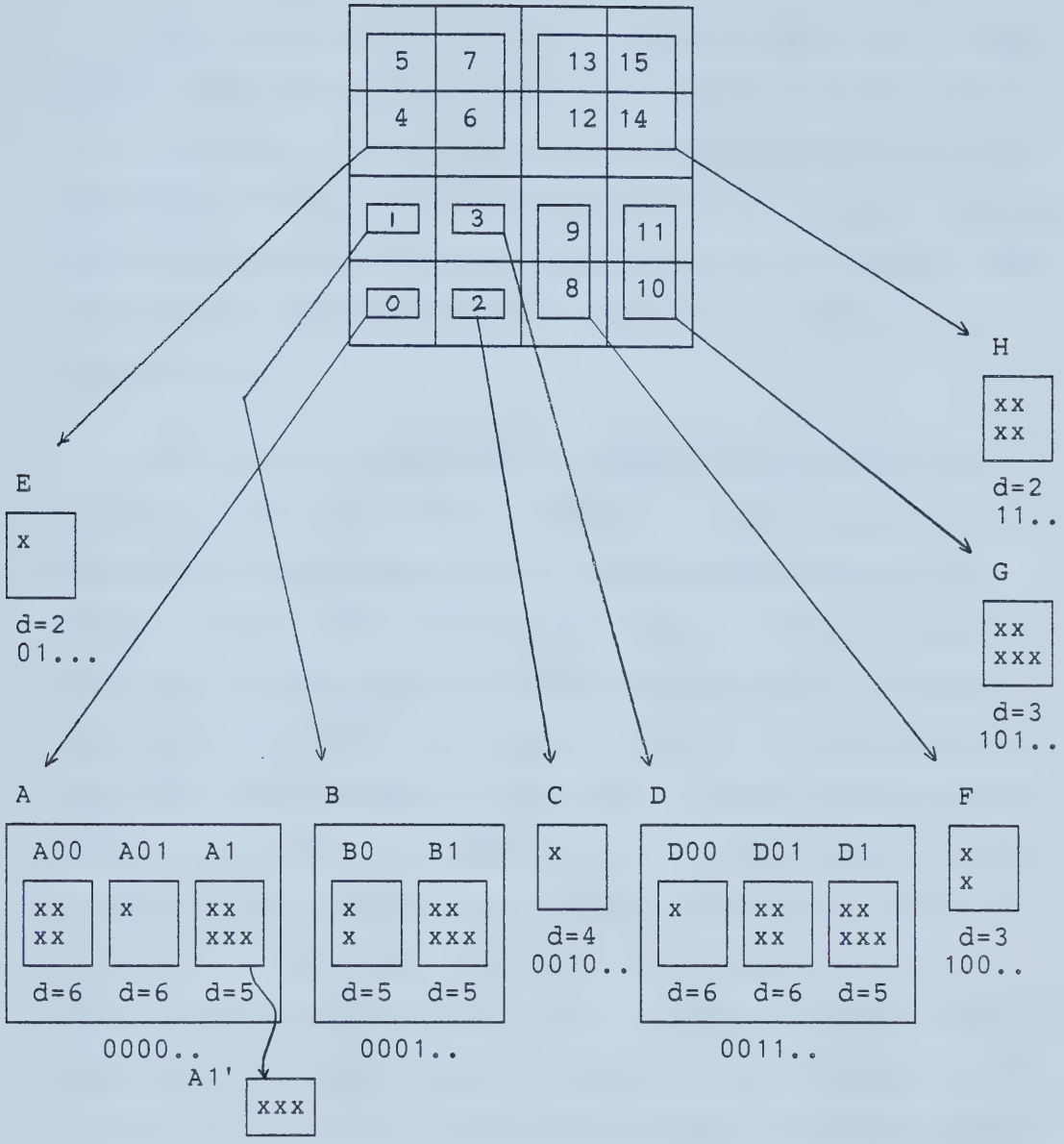


Figure 4.4. Correspondence between
Blocks and Data Buckets

4.2.3 Correspondence between Subblocks and Data Buckets

When a minimal block is subdivided into a set of subblocks, mapping by some other means is required in addition to a directory. That is, the corresponding pointer in the directory points to the starting address of a set of buckets that are physically located together, but it does not specify the correspondence between each of the subblocks and data buckets.

According to Definition 5, there exist at most two different sizes of subblocks within a single minimal block. Figure 4.5 shows examples of a subblock partition of a minimal block. The subblocks of Figure 4.5a are the same size, but the subblocks of Figure 4.5b are not. The subblocks of a minimal block are indexed in a similar manner as the cells corresponding to directory elements are indexed. In fact, when all the subblocks are the same size, they are indexed in the exactly same manner. Examples are shown in Figures 4.5a and 4.5c. However, when there exist two different sizes of subblocks, the larger subblocks have two candidates for their index. In that case, the smaller of the two is selected for the index of the subblock as shown in Figures 4.5b and 4.5d. As a result, when subblocks are of different sizes, the indexes of subblocks are not continuous.

Every subblock has an expandable data bucket associated with it. The keys of the records contained in a subblock of

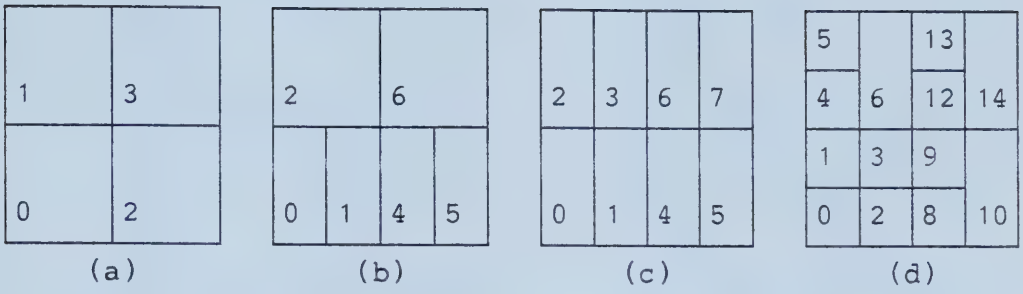
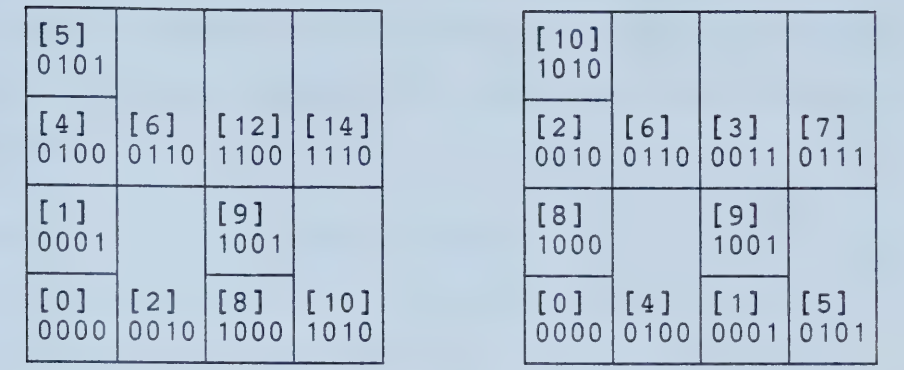


Figure 4.5. Subblocks and their Indexes

depth d have the same bit pattern in their first d bit places. More explicitly, their leftmost *maxd* bits agree with the index of the minimal block the subblock belongs to, and the next (d - *maxd*) bits agree with the index of the subblock itself. Now, it remains to show how the mapping between subblocks and buckets is made available without using a directory. The mapping is achieved essentially by numbering each of the buckets that belongs to the same minimal block in a specific way as follows.

Let a data bucket D be associated with a subblock whose index is i. Furthermore, let the maximum depth of existing subblocks be d. Then, i can be represented by a bit string S which is (d - *maxd*) bits long. Next, let k be a number represented by a bit string S' which is a reversed bit string of S. Then, k is the bucket number of D, i.e., D is the $(k + 1)th$ of the set of buckets that are contiguously located. The proof of this "reversed bit pattern" relation between i and k can be easily shown by induction (see [7] for a formal proof). Figure 4.6 illustrates the



(a) Indexes of subblocks (b) Data bucket numbers

Figure 4.6. Correspondence between Subblocks and Data Buckets

correspondence between data buckets and subblocks.

4.3 Provisions for Dynamic File Maintenance

The proposed file organization scheme allows a file structure to adapt its shape automatically to the nature, i.e., the amount and the pattern of distribution, of the data to be stored. The adaptability of the scheme is obtained mainly by a dynamic partition of the data space, which is implemented by *splitting* and *merging* mechanisms.

4.3.1 Splitting

As more and more data are inserted in a file, data buckets overflow and this results in splitting of buckets. There are four kinds of splits possible. Three of them are triggered by the insertion of a record into a fixed data

bucket which is already full. The fourth type of split occurs when a record is assigned to an expandable bucket, and the load factor exceeds the upper limit as a result of insertion.

The first three cases of splits are explained first. Let *d* denote the depth of a directory and *db* the depth of a data bucket. For the first type of split, consider the situation in which a record is assigned to a data bucket that is full and pointed to by more than one pointer of the directory, i.e., $db < d$. In this case, the overflow bucket is split to resolve the collision, and the pointers in the directory are adjusted to reflect this split. The second type of split arises when the overflow bucket is pointed to by a single pointer, and the directory has not reached its maximum yet, i.e., $db = d$ and $d < maxd$. Then, in addition to a data bucket split, refinement of the cell partition is required as well as a directory doubling. The third case occurs when the depth of the directory has reached its maximum already, i.e., $db = d = maxd$. Then, the overflow bucket is split into two expandable buckets, numbered 0 and 1, bucket 1 being allocated after bucket 0. This is called a *linear* bucket split after *linear* hashing.

The fourth type of split is somewhat different from the first three. Suppose a record is assigned to a data bucket of depth $db > d$. Then, the record is first inserted into the bucket, or if necessary, into its overflow field, and

the overall load factor of the set of buckets that are associated with the same minimal block is calculated and checked against the predefined range. If the load factor exceeds the upper limit, a linear bucket split is triggered, i.e., a new bucket is allocated at the end of the existing buckets of the set, and the bucket designated by the variable *next to split* is split into two. If the load factor still exceeds the upper limit, the splitting process is repeated.

Similar to linear hashing, the following two variables are used for dynamic file maintenance: j, *split level*, and p, *next to split*. Split level, j, indicates the level of linear splits within each minimal block. Initially, j is set to 0 for every minimal block but is increased as linear bucket splits are performed so that

$$j = \text{maximum} \left\{ \begin{array}{l} \text{depths of all subblocks} \\ \text{within the minimal block} \end{array} \right\} - maxd.$$

Next to split, p, points to the bucket which is to split next. It is initially 0 for every minimal block, but is increased by one as a linear bucket split occurs. However, at the end of each cycle of linear bucket splits, p is reset to 0. That is, during the first cycle of splits, bucket 0 is split; during the second cycle, first, bucket 0, and then bucket 1 is split; and during the k-*th* cycle, buckets are split in the order of 0, 1, ..., $2^{k-1}-1$.

Therefore, at split level $j > 0$, if $p = 0$, the number of buckets associated with the same minimal block is exactly

2^j and every bucket has a depth *maxd+j*; if $p \neq 0$, however, the number of buckets is $2^{j-1}+p$ and there exist two different depths of buckets, i.e., *maxd+j* and *maxd+j*-1.

The dynamic nature of the split is best illustrated by an example. Consider the data shown in Figure 4.7a1. Let the maximum depth of a directory *maxd* = 2, the capacity of a data bucket b = 5, the capacity of an overflow field b' = 3, and the lower and upper limits of the load factor be 0.45 and 0.75, respectively. Then, the study area is partitioned into two blocks as in Figure 4.7a1, and the data are structured in a file as shown in Figure 4.7a2. Two data buckets are used to store the data: bucket *A* contains the keys that start with '0', and bucket *B* those starting with '1'.

Now, suppose the leaf node in the NE quadrant of Figure 4.7a1 is split into four leaf nodes as a result of polygon boundary refining as shown in Figure 4.7b1. Then, the record of the original leaf node is deleted from the file and four new records are inserted instead. At the insertion of new records, bucket *B* overflows and is split into two

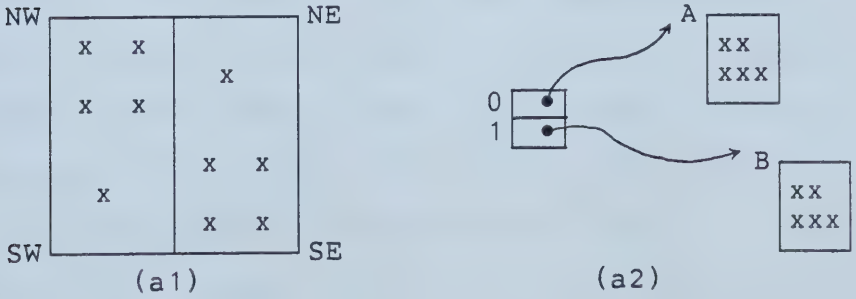


Figure 4.7a. Dynamic File Maintenance: Initial State

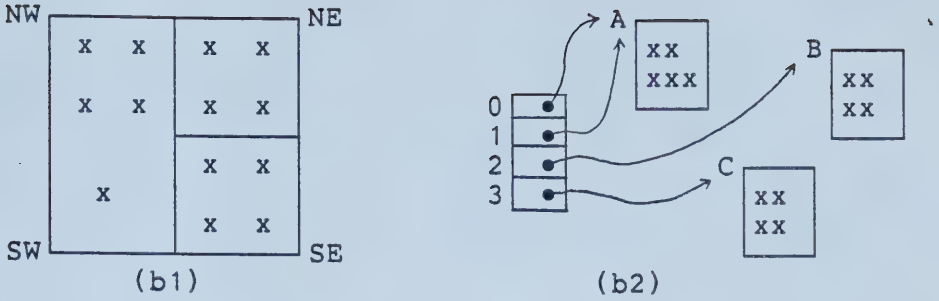


Figure 4.7b. Dynamic File Maintenance: Splitting

buckets, *B* and *C*. Bucket *B* now contains the keys starting with "10", and Bucket *C* those starting with "11", as shown in Figure 4.7b2. A directory doubling is triggered to reflect the cell partition refinement.

A directory doubling involves copying of the entire directory in such a manner that the old contents of element *i*, for $i = 0,1,\ldots,2^d-1$, where *d* is the old value of directory depth, is copied into elements *2i* and *2i+1*. Note that data buckets are not affected by a directory doubling except the overflow one: only the records in the overflow bucket are reorganized.

Next, suppose the leaf node in the SW quadrant in Figure 4.7b1 is split into four leaves as shown in Figure 4.7c1. Then, bucket *A* splits into *A* and *D* as shown in Figure 4.7c2. Notice there is no need of cell partition refinement this time. Now, suppose the top right leaf node in the SE quadrant of Figure 4.7c1 splits into four leaf nodes as in Figure 4.7d1. This time, bucket *B* overflows.

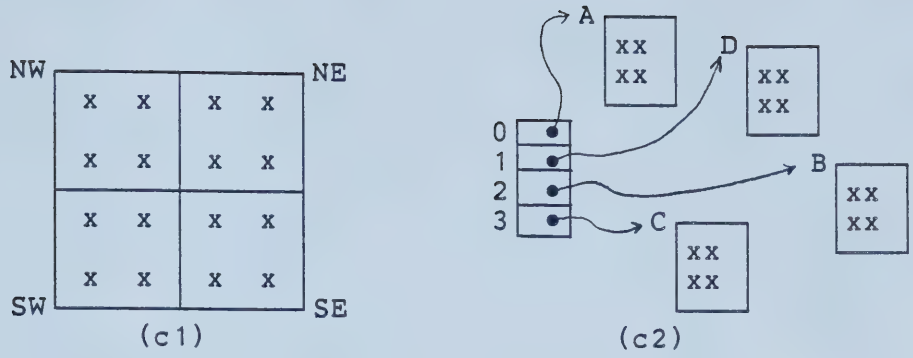


Figure 4.7c. Dynamic File Maintenance: Splitting

However, since the directory has reached its maximum depth, a linear split is applied to bucket *B*. Thus, the SE quadrant is partitioned into two subblocks and bucket *B* is split into two buckets, indexed 0 and 1. This is shown in Figures 4.7d1 and 4.7d2.

Next, suppose the top left leaf node in the SE quadrant of Figure 4.7d1 splits into four leaves as shown in Figure 4.7e1. As new records are inserted, the load factor of bucket *B*'s exceeds the upper limit and a linear bucket split

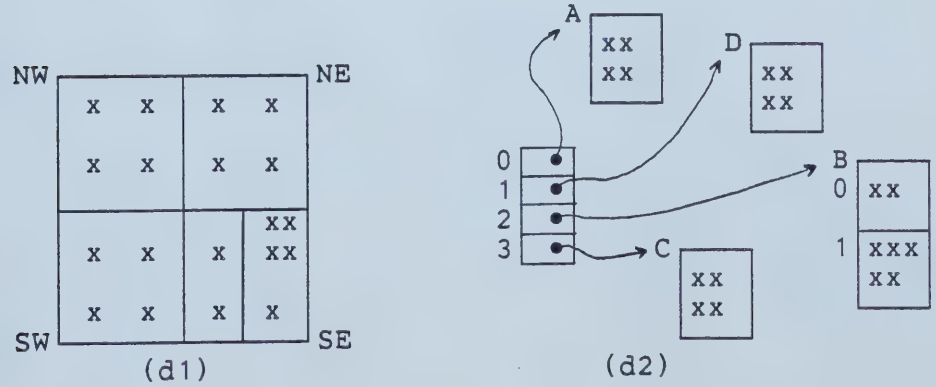


Figure 4.7d. Dynamic File Maintenance: Splitting

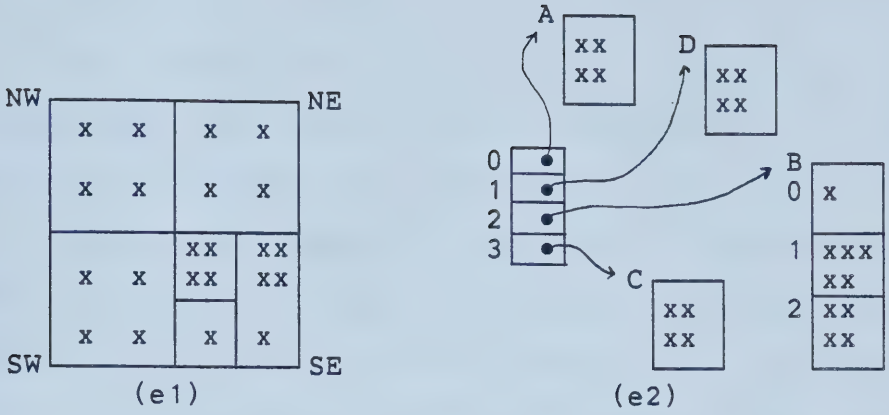


Figure 4.7e. Dynamic File Maintenance: Splitting

is triggered. Again, bucket *B0* is chosen for a split, and the resulting file structure is as shown in Figure 4.7e2. The load factor is now 10/15 = 0.67. Next, suppose the bottom right leaf node in the SE quadrant of Figure 4.7e1 splits into four as shown in Figure 4.7f1. Then, new records are inserted into *B1*, or more exactly into an overflow field attached to *B1*, as shown in Figure 4.7f2, and the load factor is recalculated. Since the load factor, 13/18 = 0.72, is within the predefined range, no split is

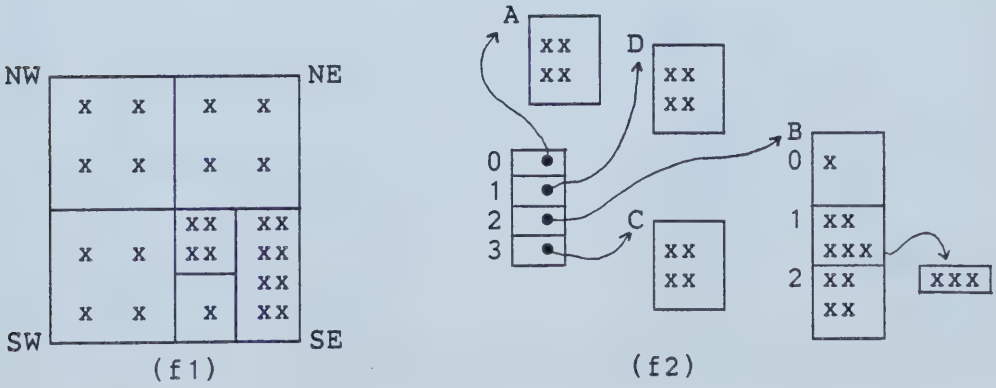
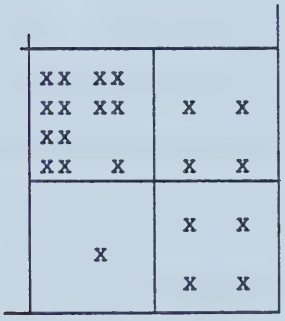


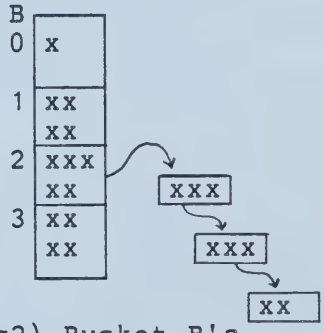
Figure 4.7f. Dynamic File Maintenance: Splitting

needed.

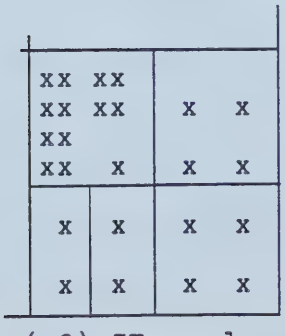
Now, suppose each of the top left, top right, and bottom left leaf nodes in the NW subquadrant of the SE quadrant in Figure 4.7f1 is split into four leaves as shown in Figure 4.7g1. Then, at some point in updating the file, the load factor of bucket *B*'s exceeds the upper limit and bucket *B1* is split into *B1* and *B3* as shown in Figure 4.7g2. However, the updated load factor is still higher than the upper limit, i.e., 22/29 = 0.76. Thus, another linear bucket split is required. Bucket *B0* is to be split next, but *B0* happens to contain a single record describing a



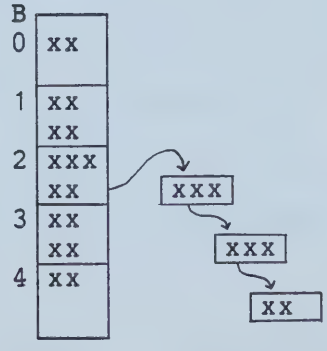
(g1) SE quadrant



(g2) Bucket B's



(g3) SE quadrant



(g4) Bucket B's after B0 splits

Figure 4.7g. Dynamic File Maintenance: Splitting

homogeneous quadtree leaf node. Therefore, the quadtree leaf node is split into four leaves of the same colour, the size of each being one fourth of the original. The result is shown in Figures 4.7g3 and 4.7g4. Now, the load factor is 25/34 = 0.74, and the splitting process stops. This is the worst possible case that may occur. In practice, however, the split of a quadtree leaf node is expected to occur rarely unless the bucket capacity is very small.

4.3.2 Merging

Merging is the inverse of splitting and results from deletions of records. Its purpose is to achieve an efficient use of memory space. Merging can occur at two levels: data bucket merging and directory merging. Of main interest is data bucket merging.

There are two kinds of data bucket merging, i.e., merging of expandable buckets and merging of fixed buckets. Merging of expandable buckets is performed whenever the overall load factor of a set of buckets that are associated with the same minimal block falls below the lower limit as a result of deletions. In this case, the candidates for merging are a pair of buddies that have been split most recently and do not neccessarily include the bucket with the lowest occupancy.

Merging of fixed buckets may take place if the total number of records contained in a pair of fixed buckets fits

into one bucket. A decision to be made here is about the threshold that determines at what bucket occupancy merging is actually triggered. Setting the threshold too low will result in a waste of memory space, whereas setting it too high will lead to poor performance, particularly in a dynamic environment. As for the candidates for merging, the simplest way is to restrict them to a pair of buddies. Then, the shape of a block that corresponds to a data bucket is always square or rectangular with the length in the y-axis direction being twice as many units long as that in the x-axis direction. Note that the merging of SW and SE quadrants or NW and NE quadrants is not possible as it violates the first requirement of Definition 7. Obviously, merging is a recursive process, proceeding upward from the leaves of the partition tree to the root node.

For an illustration of merging, suppose all the leaf nodes on the NW and NE subquadrants of Figure 4.7g3 are changed to the same colour. Then, each of the NW and NE subquadrants will contain a single record as in Figure 4.8a1, and the load factor of bucket *B*'s will drop below the lower limit, i.e., 10/25 = 0.4, triggering merging of expandable buckets. Since the pair of buddies that have been split last is *B0* and *B4*, *B4* is merged into *B0* as shown in Figure 4.8a2. However, the four records in bucket *B0* are now the same colour and thus should be merged into one record. This triggers another expandable bucket merging, and the result is as shown in Figures 4.8a3 and 4.8a4.

Now, suppose the four leaves in the NW quadrant of Figure 4.8a3 are changed to the same color and merged into a single record as in Figure 4.8b1, and similarly are the four leaves in the SW quadrant. Then, buckets *A* and *D*, which are buddies of each other, are merged into one as shown in Figure 4.8b2. Next, suppose all the leaves in the SE quadrant of Figure 4.8b1 are changed to the same colour. Then, bucket *B*'s will merge into a single fixed bucket with one record in it. This results in another bucket merging: this time, buckets *B* and *C* are merged. This in turn triggers directory merging, and the final result is as shown

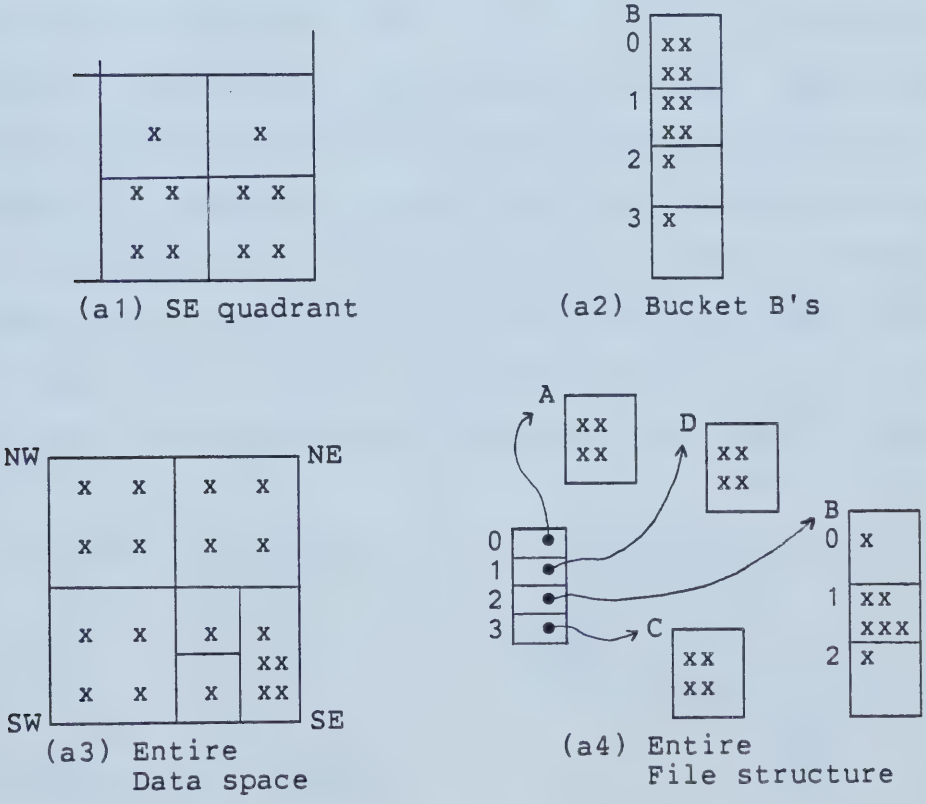


Figure 4.8a. Dynamic File Maintenance: Merging

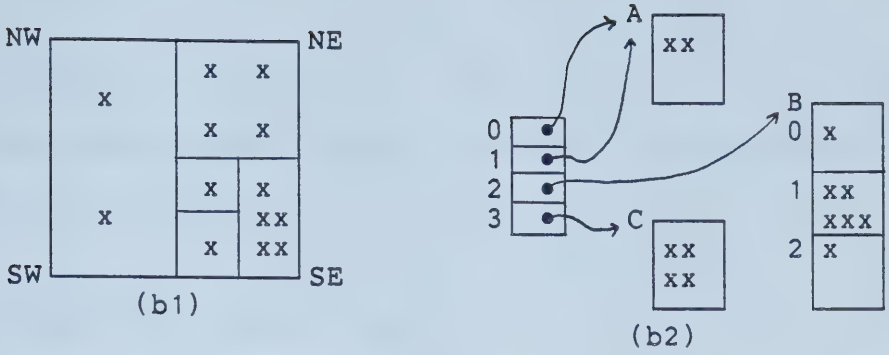


Figure 4.8b. Dynamic File Maintenance: Merging

in Figure 4.8c.

Directory merging can be triggered when every pair of pointers $2i$ and $2i+1$, for $i = 0,1,\ldots,2^{d-1}-1$, where d is the depth of a directory, have identical values. The merging process involves copying the old contents of element $2i$ into element i. Directory merging causes shrinking of the directory size into a half. Nevertheless, it may not be cost effective to merge a directory as soon as possible because a file may later expand to its previous size. Thus, directory merging is recommended only when the file itself is known to

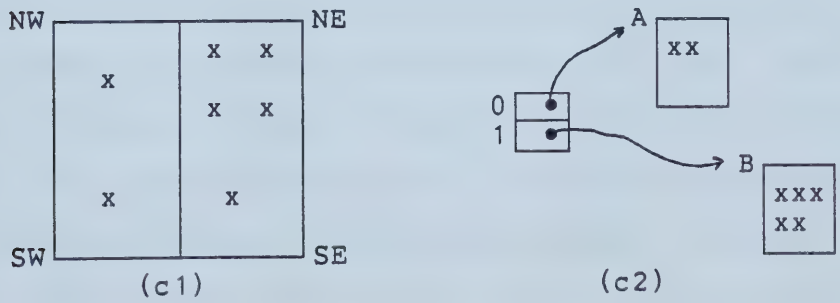


Figure 4.8c. Dynamic File Maintenance: Merging

be shrinking.

Detailed algorithms for file maintenance are presented in **Algorithms Access, Insert** and **Delete** in Appendices B, C and D, respectively.

4.4 Support for Spatial Queries

This section describes how the proposed file scheme supports various spatial queries, i.e., point-in-polygon, set operations and range search.

4.4.1 Point-in-Polgyon

The point-in-polygon query is solved using **Algorithm Access.** That is, given a point (x,y) on the study area, **Algorithm Access** retrieves a data bucket that contains the record describing the given point. An example will be given to illustrate how the algorithm works using Figure 4.9.

Suppose the study area $\mathbf{U} = [0,2^4)^2$ and the query point (x,y) = (9,15). Furthermore, suppose the depth of a directory $d = 2$ and the predefined maximum depth of a cell partition $maxd = 2$, as well. Then, the study area is partitioned into 2^2 cells as shown in Figure 4.9. The data bucket containing the desired record is determined as follows:

(1) Using function S, compute the key for the point (x,y).
In this example, key = 215 ("11010111" in binary).

(2) With the leftmost d bits of the key, determine the index

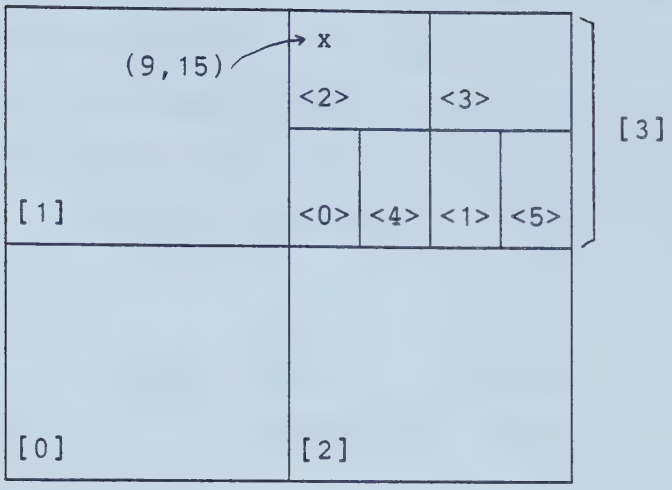


Figure 4.9. Point-in-Polygon

i of the cell that contains the given point. Thus, *i* = 3("11").

(3) Read split level *j* and next bucket to split *p* from the directory element *i*. (See Appendix A for the data structure implementing a directory.) Here, *j* = 3 and *p* = 2.

(4) If *j* = 0, the wanted bucket can be directly located by following the pointer of the directory element *i*. Since *j* > 0 in this example, the address of the bucket should be interpolated as described next.

(5) Determine the subblock index *isub* with *j*-1 bits of the key (if *p* = 0, *j* bits of the key instead of *j*-1) starting at bit position *maxd*+1 from the left. Thus, *isub* = 1("01").

(6) Calculate bucket number *bnum* by reversing the bits of

isub. Hence, *bnum* = 2("10"). (If *bnum* > *p*, the bucket number should be recalculated because the bucket has been split.) Now, the wanted bucket is the 3*rd*, i.e., (*bnum*+1)*th*, of the buckets that are contiguously located starting at the address pointed to by the directory element *i*.

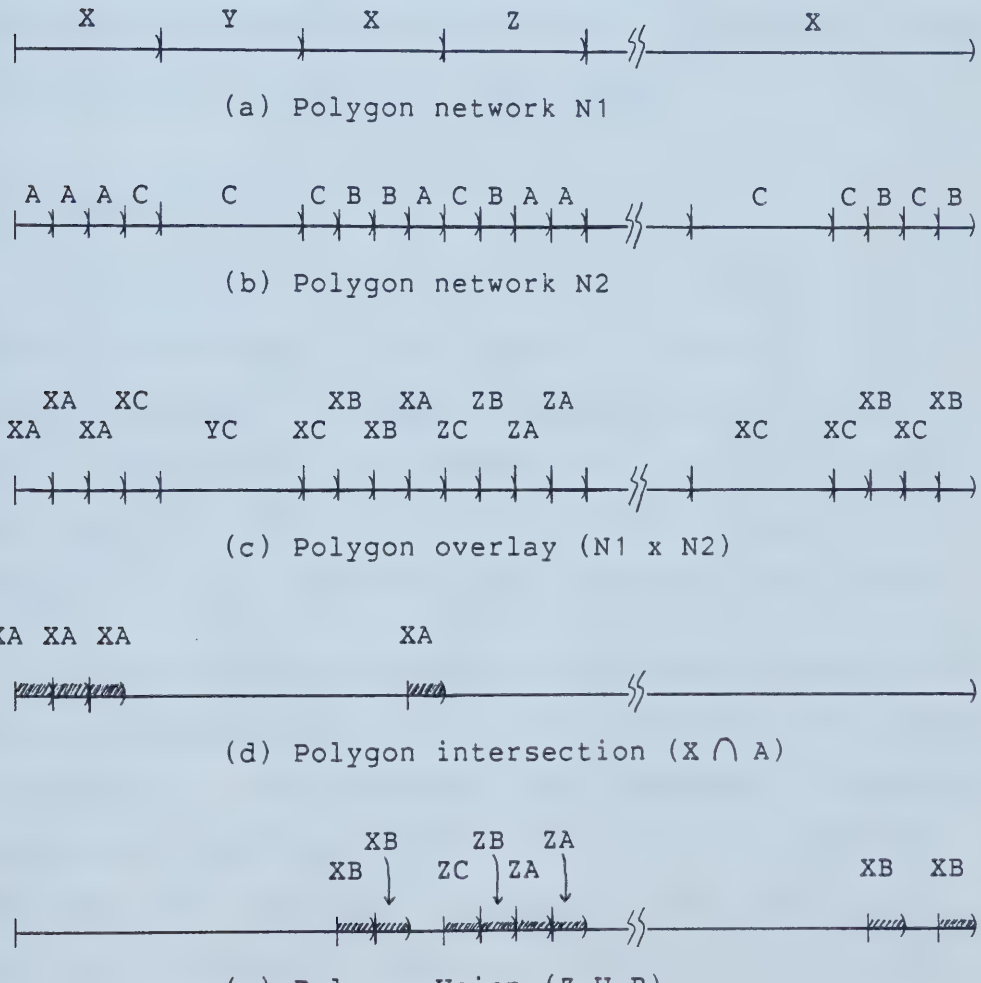
Note that the operations required to implement this algorithm are basically bit shifting and bit reversing.

4.4.2 Set Operations

The polygon overlay problem requires a simultaneous traversal of two or more quadtrees. Since a quadtree is effectively a totally partitioned 1-dimensional line, the result of overlaying two quadtrees is equivalent to that of overlaying two 1-dimensional lines as illustrated in Figure 4.10c. Thus, polygon overlay requires all the records in the file to be retrieved in sequential order by their keys. This is easily accomplished by the proposed scheme as the address of the next bucket (or next set of buckets) is stored in the directory. In case more than one bucket, i.e., a set of expandable buckets, is pointed to by a directory element, the order among these buckets is determined using the reverse bit pattern relationship between the subblock index and the bucket number. **Algorithm SequenRetrieve** in Appendix E describes in detail how each data bucket in the file is retrieved in order so that

sequential processing of records is possible without referencing the same bucket more than once.

Polygon union or intersection operations also require sequential processing of data. In contrast to polygon overlay, however, these operations do not need to retrieve all the buckets in the file. As shown in Figures 4.10d and



(a) Polygon network N1

(b) Polygon network N2

(c) Polygon overlay (N1 x N2)

(d) Polygon intersection (X ∩ A)

(e) Polygon Union (Z U B)

Figure 4.10. Set Operations

4.10e, most of the records in the file except the ones describing the query polygons can be ignored. Given the bounding rectangles of the polygons involved, the proposed scheme first identifies regions of interest in the study area by computing the union or intersection of the bounding rectangles. The blocks or subblocks intersecting those regions of interest are then determined by the range search algorithm which is described in the following section and the data bucket associated with each of the relevant blocks or subblocks is retrieved for processing.

4.4.3 Range Search

Orthogonal range search is first discussed. Given two points, (x1,y1) and (x2,y2), where $x1 \leq x2$ and $y1 \leq y2$, specifying a query rectangle, the proposed file scheme is able to retrieve every data bucket that contains the records describing the quadtree leaf nodes which overlap the query rectangle without retrieving any irrelevant data buckets.

An algorithm for identifying the relevant data buckets for a given query rectangle is described using the example of Figure 4.11. The algorithm takes advantage of the Morton sequence that has been used for indexing the blocks and subblocks. For the sake of simplicity, the depth of a directory is assumed to be an even number in the example.

Suppose a directory has depth 4 which is the same as the predefined maximum depth. Suppose also that the shaded

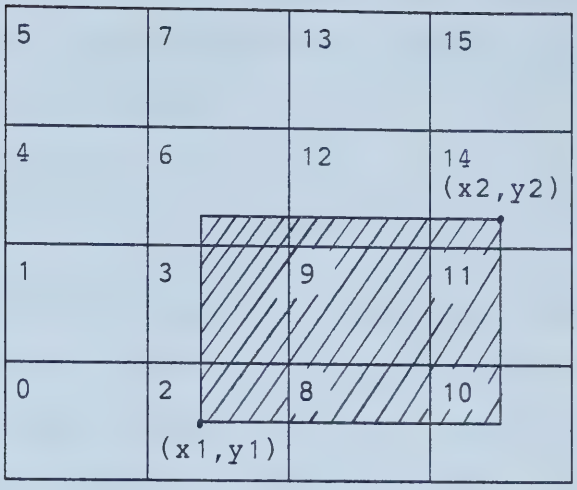


Figure 4.11. Range Search

area in Figure 4.11 is the query rectangle. Determination of the cells that intersect the query rectangle is done as follows:

(1) Using **Algorithm Access**, determine the indexes of cells in which (x1,y1) and (x2,y2) are contained. In this example, they are 2 and 14.

(2) Decompose the bit pattern of these indexes into their x- and y-components. Let the x-component of the lower index be the lower limit of x and the x-component of the higher index be the upper limit of x. Similarly, determine the lower and upper limits of y.

Lower index	0010 (2) xyxy	xlow: 01 (1) ylow: 00 (0)
Higher index	1110 (14) xyxy	xhigh: 11 (3) yhigh: 10 (2)

(3) For each of x and y, create a set of numbers which contains all the integers that are between the lower and upper limits inclusive.

x: {01(1), 10(2), 11(3)}

y: {00(0), 01(1), 10(2)}

(4) Obtain a cross product of these sets, where each member of the resulting set is an integer produced by interleaving the x and y components. This set then indicates the cells that overlap the query rectangle. In this example, the following cells overlap the query rectangle: 2, 3, 6, 8, 9, 10, 11, 12 and 14.

y \ x	01 (1)	10 (2)	11 (3)
00 (0)	0010 (2)	1000 (8)	1010 (10)
01 (1)	0011 (3)	1001 (9)	1011 (11)
10 (2)	0110 (6)	1100 (12)	1110 (14)

Next, suppose some of the cells have been further subdivided, e.g., cells 9 and 12. As for cell 9, every expandable bucket associated with it should be retrieved since the cell is completely contained in the query rectangle. Retrieval of these buckets can be done using **Algorithm SequenRetrieve** (Steps 5-6). As for cell 12, however, a decision should be made regarding which of its subblocks intersects the query rectangle because the cell is only partially contained in it. Determination of the subblocks that intersect the query rectangle is done in a

similar manner as determination of the cells intersecting the query rectangle. The bucket number corresponding to each relevant subblock is then obtained by reversing the bits of the subblock index. The detailed algorithm for orthogonal search is given in **Algorithm RangeSearch** in Appendix F.

Finally, circular range search is supported as follows. Given the center and the radius of a query circle, a bounding rectangle is first obtained. Using the algorithm described above, the blocks and/or subblocks that overlap the bounding rectangle are determined. Each of the blocks and/or subblocks thus obtained is then tested to see whether or not it overlaps the query circle. If so, its associated data bucket is retrieved. As the algorithm for circular range search is mostly identical to the one for orthogonal range search, a separate algorithm is not given.

CHAPTER 5

EVALUATION OF THE PROPOSED SCHEME

In this chapter, the proposed file organization scheme is evaluated and compared with other file organization schemes in terms of the variety of spatial queries supported and the access efficiency for single record retrieval.

The object used for comparison is a B-tree that has been proposed and implemented for representing a polygon network by a linear quadtree [1,2,27,29]. In particular, the B^+-tree version is considered. As defined in [8], in a B^+-tree, all the keys reside in terminal nodes and the upper level of the tree consists merely of an index. Furthermore, terminal nodes are linked together left-to-right to facilitate sequential processing of the file. In addition to a B^+-tree, the EXCELL method is also used for comparison. Although the EXCELL method was originally used for representing a polygon network in vector format [31], the method is also useful for representing a polygon network by a quadtree.

5.1 Types of Spatial Queries Supported and Their Efficiency

A point-in-polygon query with raster representation is essentially a problem of retrieving a single record from a file. Consequently, the efficiency of solving a point-in-polygon problem is equivalent to the access efficiency of

single record retrieval.

Polygon overlay requires sequential processing of records. With the proposed scheme and the EXCELL method, a directory is used to identify the location of the next bucket (or set of buckets). Note that the proposed scheme requires a bit reversing operation in calculating the bucket number from a subblock index. However, as the processing is carried out in main memory, its cost in time is almost negligible compared with the time required for an I/O operation. As for a B^+-tree, it is particularly convenient for sequential processing because terminal nodes are linked in order. Note that the cost for polygon overlay is also related to the number of buckets in a file because polygon overlay requires to retrieve the entire file. That is, the more buckets exist in the file, the more I/O operations are needed.

Polygon union and intersection also require sequential processing of files to some extent. However, these operations differ from polygon overlay in that the entire file does not have to be processed. Thus, it is desirable that only those data buckets that are relevant to a query are retrieved. Given the bounding rectangles of the polygons involved, the proposed scheme as well as the EXCELL method can identify relevant buckets by first computing the union or intersection of those bounding rectangles. This is not possible with a B^+-tree as there is no implied

correspondence between data buckets and regions in the study area. Consequently, the whole file will be retrieved even though the query can be answered with the information from a small part of the file.

Finally, range search is unique to cell methods. With a B^+-tree, virtually the entire file has to be retrieved for range search. Indeed, it has been reported [27] that windowing of a region of a map is very expensive with a B^+-tree structure. Therefore, unless queries are limited to point-in-polygon and polygon overlay, a B^+-tree organization is not a good choice.

5.2 Access Efficiency and Storage Utilization

When data reside in external storage, the number of accesses to secondary storage is a dominating factor in the total time needed to process a query. Thus, in the following discussion, the prime consideration is given to the access efficiency, i.e., the number of disk accesses required for retrieving each record in a file, although storage utilization is taken into consideration as well.

The B^+-tree maintains consistent performance both in storage and access time as its structure is not affected by the pattern of data distribution. Its buckets are more or less uniformly filled (each bucket being at least half full) even with non-uniformly distributed data, and every record in the file can be retrieved with $O(\log n)$ disk accesses,

where n is the number of records in the file.

In contrast, cell methods are very sensitive to the pattern of data distribution. Both with the proposed scheme and the EXCELL method, the best case occurs when data are distributed uniformly over the study area. Then, the proposed scheme requires one disk access to locate a record with a given key while the EXCELL method requires two disk accesses, both with $O(n)$ storage.

The worst case occurs when data are distributed non-uniformly over the study area. As with all hashing schemes, the worst case performance of the proposed scheme as well as that of the EXCELL method is intolerable. With the EXCELL method, the directory will become large and unwieldly and the goal of two disk accesses for retrieving a record cannot be achieved in general. On the other hand, with the proposed scheme, there will be a long chain of overflow fields as well as many underflow buckets. Thus, the proposed scheme may take $O(n)$ disk accesses for single record retrieval in the worst case. Moreover, frequent quadtree leaf node splits will incur a greater storage requirement.

The poor performance of cell methods is due to their extreme sensitivity to the existence of a random cluster of data. However, the probability of this worst case happening in practical data is expected to be exceedingly low. Moreover, as a hybrid of extendible hashing and linear

hashing, the proposed scheme allows its structure to be considerably adaptable to the nature of data. As the directory of the proposed scheme divides the study area into a coarse grid, any non-uniformity of data distribution affects the file structure only within a grid cell rather than over the entire study area.

Since a worst case analysis does not provide meaningful conclusions, an expected case should be studied. Unfortunately, it is hard to theoretically establish an expected case analysis because of the difficulty in constructing a representative polygon network model. Thus, performance of each file scheme has been simulated using a set of real data.

5.2.1 Experimental Results

In order to test the feasibility of the proposed scheme and obtain some preliminary estimation of its performance, the scheme has been applied to the data shown in Figure 5.1 which is a surficial geology map of the Wabamun area in Alberta, Canada (114°-115°W and 53.5°-54°N). The map has been converted into an image of 2' x 2' unit cells, and a quadtree has been constructed from the image. The resulting quadtree consists of 53,830 leaf nodes. Then, a file has been constructed with the leaf nodes.

The proposed scheme requires the following parameters to be supplied with values:

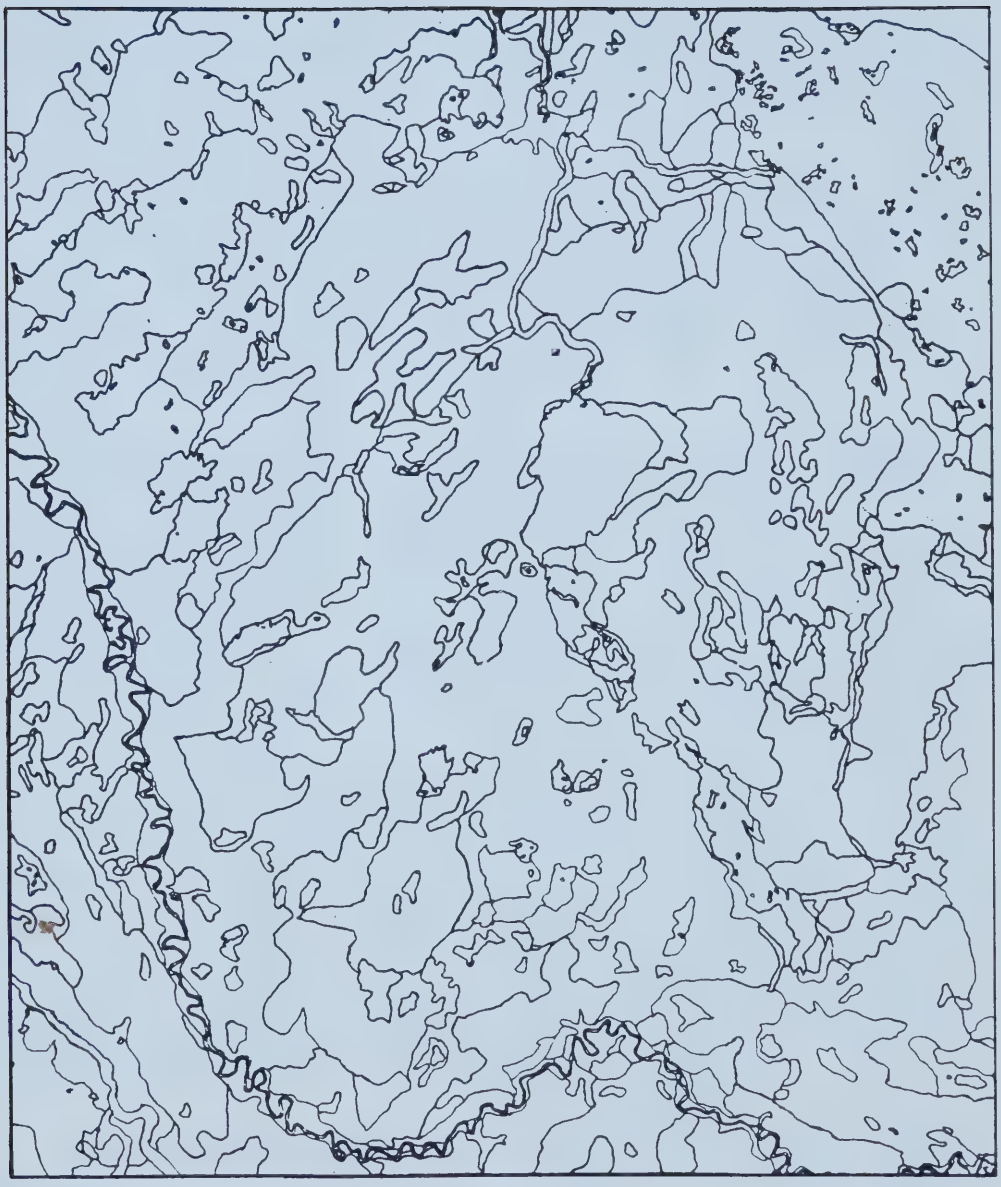


Figure 5.1. A Surficial Geology Map

a) d, the maximum depth of the directory,

b) b, the capacity of a bucket,

c) b', the capacity of an overflow field,

d) the upper and lower limits of w, the load factor, and

e) threshold in bucket occupancy that triggers merging of fixed buckets.

Three different values, i.e., 2, 4 and 6, were used for the maximum depth of the directory (with the proposed scheme, the maximum depth is normally small so that the directory may reside in main memory).

As for the capacity of a bucket, b = 60 was chosen, a bucket being assumed to be 512 bytes long and each record describing a quadtree leaf is 8 bytes long (key: 24 bits; size, or level, of the leaf: 12 bits; and color information: 28 bits). The capacity of an overflow field was fixed to b'= 30 arbitrarily. However, for a real implementation of the scheme, an experiment is required to choose an optimal value for b relevant to the environment. Although b should be >> 1, it cannot be indefinitely large as the processing cost within a bucket depends on b. Also, an experiment is required with various ratios between b and b'.

Two different values were used for the upper limit of the load factor, i.e., $w \leq 0.65$ and $w \leq 0.70$. The lower limit on w and the threshold that triggers merging of fixed buckets were not specified as the experiment did not include any deletion operations.

The following measurements were then taken to estimate the performance of the scheme:

a) the total number of quadtree leaf node splits,

b) the total number of buckets,

c) the total number of overflow fields,

d) the total number of buckets with no overflow fields attached to them, and

e) lengths of overflow chains.

From these measurements, the following have been calculated: (1) Qov, the average number of I/O operations required to access an overflow field, (2) Qr, the expected number of I/O operations required to retrieve a single record, and (3) W, actual storage utilization ratio. The results are shown in Table 1 and Table 2. Note that the following formulas have been used for the tables:

$$A_i = (\Sigma L_k, \text{ for } i-1 \le k \le l), \text{ for } 2 \le i \le l+1,$$

$$Qov = (\Sigma i A_i)/M', \text{ for } 2 \le i \le l+1,$$

$$Qr = [bM+(Qov)b'M']/(bM+b'M'), \text{ and}$$

$$W = n/(bM+b'M'),$$

where l is the length of the longest chain of overflow fields and n is the number of original quadtree leaf nodes. The formula for Qr is only approximate since it uses bM and b'M' rather than the exact count of records in each of the buckets and overflow chains.

Notice that the split of a quadtree leaf node occurs only occasionally. The longest overflow chain is length 6.

Table 1

Experimental Results for the Proposed File Scheme with an Upper Limit of the Load Factor 0.65

DESCRIPTION / MAX DEPTH OF DIRECTORY		2	4	6
No. of quadtree leaf node splits		8	7	11
No. of buckets (M)		1107	1121	1154
No. of buckets with no overflow fields attached		716	750	807
No. of overflow fields (M')		550	534	505
Length of overflow chains: "Lk" denotes the no. of overflow chains with length k.	L1	259	243	230
	L2	107	98	88
	L3	23	25	20
	L4	2	5	7
	L5	0	0	1
	L6	0	0	1
No. of I/O's required to access an overflow field: "Ai" denotes the no. of overflow fields accessible with i number of I/O's.	A2	391	371	347
	A3	132	128	117
	A4	25	30	29
	A5	2	5	9
	A6	0	0	2
	A7	0	0	1
Avg no. of I/O's to access an overflow field (Qov)		2.3418	2.3801	2.4257
Expected no. of I/O's to access a record (Qr)		1.2670	1.2655	1.2559
Storage utilization ratio (W)		0.6492	0.6464	0.6379

Table 2

Experimental Results for the Proposed File Scheme with an Upper Limit of the Load Factor 0.70

DESCRIPTION / MAX DEPTH OF DIRECTORY		2	4	6
No. of quadtree leaf node splits		2	4	4
No. of buckets (M)		957	963	1009
No. of buckets with no overflow fields attached		540	560	625
No. of overflow fields (M')		652	650	593
Length of overflow chains: "Lk" denotes the no. of overflow chains with length k.	L1	250	232	235
	L2	116	115	110
	L3	37	38	23
	L4	11	16	13
	L5	3	2	1
	L6	0	0	2
No. of I/O's required to access an overflow field: "Ai" denotes the no. of overflow fields accessible with i number of I/O's.	A2	417	403	384
	A3	167	171	149
	A4	51	56	39
	A5	14	18	16
	A6	3	2	3
	A7	0	0	2
Avg no. of I/O's to access an overflow field (Qov)		2.4954	2.5308	2.5008
Expected no. of I/O's to access a record (Qr)		1.3800	1.3863	1.3409
Storage utilization ratio (W)		0.6993	0.6966	0.6872

Thus, 7 disk accesses are required to retrieve an overflow field in the worst case. However, the majority of the buckets have no overflow fields attached to them. As expected, there is a tradeoff between storage utilization and the expected number of disk accesses required to retrieve a record in the file. As for the various values of a directory depth, test results do not show much difference. But the difference is expected to be more distinguishable with non-uniformly distributed data.

Next, the same data have been used to estimate the performance of a B^+-tree and the EXCELL method. For both schemes, a bucket capacity b=60 were used. With the EXCELL method, the directory has grown to a depth of 12. The depth of a B^+-tree turned out to be 2. (It has been assumed that the data arrive in sequential order by their keys.)

Table 3 compares the experimental results of the three file schemes. It shows that the proposed scheme performs better than either a B^+-tree or the EXCELL method in the expected number of disk accesses required to retrieve a record in a file, with a higher storage utilization ratio.

From these test results, one is tempted to characterize the proposed scheme as a method of locating a record with a given key normally in one disk access or at most two, with $O(n)$ storage. Although a formal proof cannot be given, it is conjectured that with the proposed scheme the expected number of disk accesses required for locating the record

Table 3

Summary of Experimental Results

TYPE OF FILE SCHEMES \ DESCRIPTION			Expec.no. of I/O's to access a record	No. of buckets	No. of overflow fields	Storage utilization ratio *
B^+-tree			2.0000	1794	N/A	0.5001
EXCELL			2.0000	1413	N/A	0.6349
Proposed Scheme	w≤0.65	d=2	1.2670	1107	550	0.6492
		d=4	1.2655	1121	534	0.6464
		d=6	1.2559	1154	505	0.6379
	w≤0.70	d=2	1.3800	957	652	0.6993
		d=4	1.3863	963	650	0.6966
		d=6	1.3409	1009	593	0.6872

Note: *Space for index or directory being ignored.

with a given key is constant, either one or two, irrespective of the file size, while that of B-trees or the (hierarchical) EXCELL method is logarithmically growing with the size of a file.

CHAPTER 6

CONCLUDING REMARKS

In most geometric databases, I/O operations are the bottleneck of their performance due to the large volume of data that should be handled. The proposed file organization scheme is an adaptive cell method which attempts to minimize the number of disk accesses in performing spatial queries by allowing a query to be decomposed into a set of subqueries within cell restrictions.

The scheme has been developed based on interpolation hashing, a k-dimensional version of linear hashing, and extendible hashing. As a hybrid of these two hashing schemes, the proposed scheme intends to combine the best features of both. The scheme retains the advantage of linear hashing, i.e., the mapping between data buckets in physical storage and regions in the search space is interpolated rather than stored. Thus, in the best case, the scheme achieves the goal of one disk access for single record retrieval, with $O(n)$ storage. Its main problem is the somewhat deteriorating performance in access time when the data are non-uniformly distributed.

By applying extendible hashing, however, the worst case performance is improved. Since a small directory at the highest level allows the search space to be divided into a coarse grid, any random cluster of data affects the file

structure only within a grid cell rather than the entire file structure. Therefore, the sensitivity of the scheme to the data distribution is significantly alleviated.

Another important feature of the proposed scheme is attributed to linear hashing. Here, a compromise between space and time can be obtained by controlling the load factor. To achieve an optimal worst case performance in the number of disk accesses required for retrieving a record in the file, the load factor must inevitably be low. On the other hand, too conservative storage utilization deteriorates the expected performance in access time. Between these two extremes, the implementer is allowed to choose a desirable storage utilization ratio.

While a theoretical analysis is difficult, experimental results with a set of real data show that the proposed scheme is superior to a B^+-tree both in access efficiency and storage utilization. Additionally, the scheme is comparable to the EXCELL method which was originally proposed for representing a polygon network by vectors.

Even though the experimental results are limited to the chosen test data, they are expected to hold for a large class of geometric data, especially for map data. How general the results are, however, is a matter of further research. Of course, it is possible to construct a set of data for which these results do not hold. More empirical work is needed to determine the practical validity of this

work.

In addition, it should be noted that the tests presented in Chapter 5 do not investigate the proposed file scheme extensively. To conclusively establish the practicality of the proposed scheme, it should be implemented on a system, and the total time to build a file as well as the total running time in answering various queries should be measured. Furthermore, the expected time to insert or delete a single record in a file and the changing structure of the file as updates occur should be investigated.

Another area of investigation is the possibility of using M'/M, i.e., the ratio of the number of overflow fields to the number of primary buckets, in computing the load factor. If the results of using this simpler formula are close enough to those of using the original $N/(bM+b'M')$, i.e., the ratio of the number of records to the number of slots available, the nuisance of counting the number of records may be eliminated.

As an extension, the proposed scheme may be applied to representing a polygon network by a DF-expression. The extension of the proposed scheme to higher dimensions may also be considered, but its potentiality is unknown at this time. Still another issue concerns a polygon network encoded in vector format. As mentioned earlier, for certain operations vector representation is more convenient. Thus, two questions arise: (a) Is the proposed scheme applicable

to vector represented data as well? (b) If so, what is the most efficient format of vector encoding?

BIBLIOGRAPHY

[1] Abel,D.J. "A B$^+$-Tree Structure for Large Quadtrees", *Computer Vision, Graphics, and Image Processing*, V.27(1984): 19-31.

[2] Abel,D.J. & J.L.Smith. "A Data Structure and Algorithm Based on a Linear Key for a Rectangle Retrieval Problem", *Computer Vision, Graphics, and Image Processing*, V.24(1983): 1-13.

[3] Bayer,R. & E.McCreight. "Organization and Maintenance of Large Ordered Indexes", *Acta Informatica*, V.1 (1972): 173-189.

[4] Bentley,Jon Louis & J.H.Friedman. "Data Structures for Range Searching", *Computing Surveys*, V.11,No.4(Dec. 1979): 397-409.

[5] Bentley,Jon Louis & D.F.Stanat. "Analysis of Range Searches in Quad Trees", *Information Processing Letters*, V.3,No.6(Jul.1975): 170-173.

[6] Bentley,J.L., B.W.Weide, & A.C.Yao. "Optimal Expected-Time Algorithms for Closest Point Problems", *ACM Trans. Mathematical Software*, V.6,No.4(Dec.1980): 563-580.

[7] Burkhard,Walter A. "Interpolation-Based Index Maintenance", *BIT*, V.23(1983): 274-294.

[8] Comer,Douglas. "The Ubiquitous B-Tree", *Computing Surverys*, V.11,No.2(Jun.1979): 121-137.

[9] Davis,Wayne A. & C.P.Gangadharaiah. "Data Structures for Spatial Data Processing", Technical Report TR 85-7, Dept. of Computing Science, Univ. of Alberta, 1985.

[10] Davis,Wayne A. & X.Wang. "A New Approach to Linear Quadtrees", *Proceedings, Graphics Interface*, Montreal, May 1985, pp.195-202.

[11] Fagin,R., J.Nievergelt, N.Pippenger & H.R.Strong. "Extendible Hashing – A Fast Access Method for Dynamic Files", *ACM Trans. Database Systems*, V.4, No.3(Sep.1979): 315-344.

[12] Friedman,J.H., J.L.Bentley & R.A.Finkel. "An Algorithm for Finding Best Matches in Logarithmic Expected Time", *ACM Trans. Mathematical Software*, V.3,No.3 (Sep.1977): 209-226.

[13] Gargantini,Irene. "An Effective Way to Represent Quadtrees", *Commun. ACM*, V.25,No.12(Dec.1982): 905-910.

[14] Hinrichs,K. & J.Nievergelt. "The Grid File: A Data Structure Designed to Support Proximity Queries on Spatial Objects", a paper presented at the Workshop on Graphtheoretic Concepts in Computer Science, Osnabruck, June, 1983.

[15] Kawaguchi,Eiji & Tsutomu Endo. "On a Method of Binary-Picture Representation and its Appication to Data Compression", *IEEE Trans. Pattern Analysis and Machine Intelligence*, V.2,No.2(Jan.1980): 27-35.

[16] Kawaguchi,Eiji, Tsutomu Endo & Jun-ichi Matsunaga. "Depth-First Picture Expression Viewed from Digital Picture Processing", *IEEE Trans. Pattern Analysis and Machine Intelligence*, V.5,No.4(Jul.1983): 373-384.

[17] Klinger,Allen & C.R.Dyer. "Experiments on Picture Representation Using Regular Decomposition", *Computer Graphics and Image Processing*, V.5(1976): 68-105.

[18] Knuth,Donald E. *Sorting and Searching, The Art of Computer Programming*, V.3, Addison-Wesley, Reading, Mass., 1973.

[19] Larson,Per-Ake. "Dynamic Hashing", *BIT*, V.18(1978): 184-201.

[20] Litwin,Witold. "Linear Hashing: A New Tool for File and Table Addressing", *Proceedings, 6th*

International Conference on Very Large Data Bases, 1980, pp.212-223.

[21] Litwin,Witold. "Virtual Hashing: A Dynamically Changing Hashing", *Proceedings, 4th International Conference on Very Large Data Bases*, 1978, pp.517-523.

[22] Nievergelt,J., H.Hinterberger & K.C.Sevcik. "The Grid File: An Adaptable, Symmetric Multikey File Structure", *ACM Trans. Database Systems*, V.9,No.1(Mar. 1984): 38-71.

[23] Nievergelt,J. & F.P.Preparata. "Plane-Sweep Algorithms for Intersecting Geometric Figures", *Commun. ACM*, V.25,No.10(Oct.1982): 739-747.

[24] Nortey,Michael S. "Point-in-Polygon Algorithms for Geographic Information Systems", M.Sc. Thesis, Dept. of Computing Science, Univ. of Alberta, 1982.

[25] Peuquet,Donna J. "A Conceptual Framework and Comparison of Spatial Data Models", Background Material to Workshop W2: Data Structures on Spatial Data Handling, International Symposium on Spatial Data Handling, Aug.1984, Zurich, Switzerland.

[26] Rosenfeld,A., H.Samet, C.A.Shaffer & R.E.Webber. "Application of Hierarchical Data Structures to Geographical Information Systems", TR-1197, Computer Science Dept., Univ. of Maryland, 1982.

[27] Rosenfeld,A., H.Samet, C.A.Shaffer & R.E.Webber. "Application of Hierarchical Data Structures to Geographical Information Systems, Phase 2", TR-1327, Computer Science Dept., Univ. of Maryland, 1983.

[28] Samet,Hanan. "The Quadtree and Related Hierarchical Data Structures", *Computing Surveys*, V.16,No.2(Jun. 1984): 187-260.

[29] Samet,H., A.Rosenfeld, C.A.Shaffer & R.E.Webber. "A Geographic Information System Using Quadtrees", *Pattern Recognition*, V.17,No.6(1984): 647-656.

[30] Tamminen,Markku. "Order Preserving Extendible Hashing and Bucket Tries", *BIT*, V.21(1981): 419-435.

[31] Tamminen,Markku. "Efficient Spatial Access to a Data Base", *Proceedings, International Conference on the Management of Data, ACM-SIGMOD*, 1982, pp.200-206.

[32] Tamminen,Markku. "The Extendible Cell Method for Closest Point Problems", *BIT*, V.22(1982): 27-41.

[33] Tamminen,Markku. "Encoding Pixel Trees", Report-HTKK-TKO-B51, Helsinki Univ. of Technology, 1983.

[34] Tamminen,Markku. "Performance Analysis of Cell Based Geometric File Organizations", *Computer Vision, Graphics, and Image Processing*, V.24(1983): 160-181.

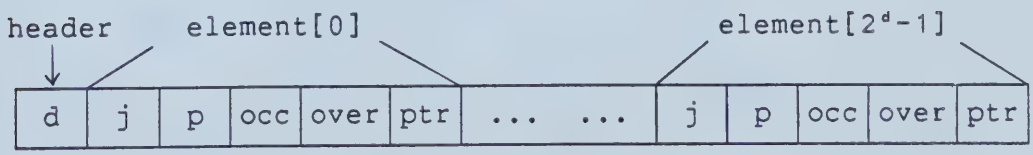
APPENDIX

Included are algorithms mentioned in Chapter 4 and data structures implementing the proposed file organization scheme. In each of the algorithms, dir[i].A denotes the field A of the (i+1)*th* element of the directory.

A. Data Structures

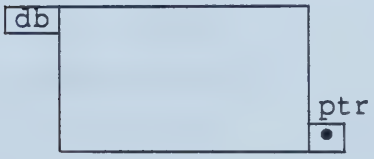
The file structure consists of two parts: a directory and data buckets. The directory has a *header* which contains the depth *d* of the directory. The header is followed by 2^d elements, indexed from 0 to 2^d-1. For the purpose of random access to the elements of the directory, the directory is best implemented as a conventional array.

Each element of the directory is a 5-tuple of <*j*,*p*,*occ*, *over*,*ptr*>, where *j* is the split level, *p* is the bucket to be split next, *occ* is the number of records contained in the data bucket (or set of expandable buckets), *over* is the number of overflow fields employed, and *ptr* is the pointer to the data bucket (or set of expandable data buckets), as illustrated below.

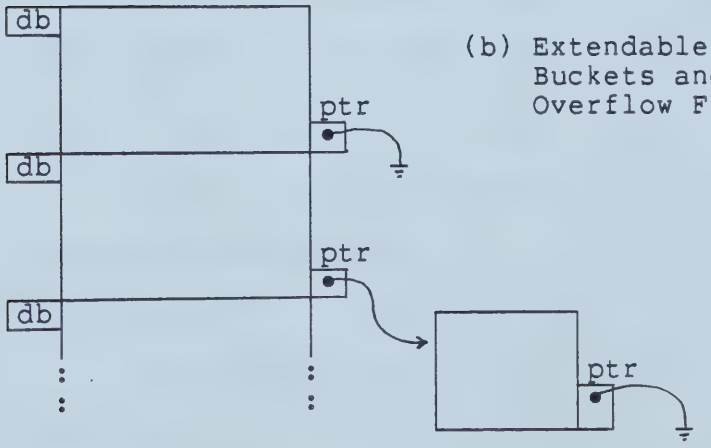


A Directory

Each data bucket or overflow field contains a set of records, (K(L),S(L),C(L)). The internal structure of a data bucket or an overflow field is left open. In addition to the set of records, each data bucket has a header that contains *db*, the bucket depth, and a trailer that contaians *ptr*, the pointer. If a bucket is an expandable one, *ptr* points to a chain of overflow fields attached to it; otherwise, it may be either ignored or used to point to the next bucket. Each overflow field has a trailer, *ptr*, which points to the next overflow field in the chain. Data sturctures for these are shown below.



(a) A Fixed Data Bucket



(b) Extendable Data Buckets and an Overflow Field

B. Algorithm Access

Input: file **F**, directory *dir* to **F**, and (x,y) ϵ **U**, where **U** = $[0,2^n)^2$ is the study area.

Output: data bucket D which contains (K(L),S(L),C(L)) such that (x,y) ϵ L. (The record may be in an overflow field of D.)

Step 1: #calculate key#

key $\leftarrow$ s(x,y)

Step 2: read *d*, depth of directory

Step 3: #determine index to directory#

$i \leftarrow \lfloor key \;/\; 2^{2n-d} \rfloor$

Step 4: #read pointer value#

loc $\leftarrow$ *dir*[*i*].*ptr*

Step 5: #split level 0? #

if *dir*[*i*].*j* = 0, goto Step 8

Step 6: #here, linear splits have occurred#

a. if *dir*[*i*].*p* = 0, #every bucket has been split#

1) #set subblock index with *j* bits of the *key* at position *d*+1 .. *d*+*j* from the left#

$isub \leftarrow \lfloor (key \bmod 2^{2n-d}) \;/\; 2^{2n-d-j} \rfloor$,

where *j* stands for *dir*[*i*].*j*

2) #calculate bucket no.#

$bnum \leftarrow \Sigma a_k 2^{m-1-k}$, for $0 \le k \le m-1$, where

isub is equivalent to $\Sigma a_k 2^k$, for $0 \le k \le m-1$

b. otherwise

1) #set subblock index with (*j*-1) bits of the key at position *d*+1 .. *d*+*j*-1 from the left#

isub ← $\lfloor (key \bmod 2^{2n-d}) / 2^{2n-d-j+1} \rfloor$,

where *j* stands for *dir*[*i*].*j*

2) #calculate bucket no.#

bnum ← $\Sigma a_k 2^{m-1-k}$, for $0 \le k \le m-1$, where

isub is equivalent to $\Sigma a_k 2^k$, for $0 \le k \le m-1$

3) #if bucket has been split, adjust bucket no.#

if *bnum* < *dir*[*i*].*p* and (*d*+*j*)*th* bit of *key* = "1",

bnum ← *bnum* + 2^{j-1}

Step 7: #calculate address of the bucket#

loc ← *loc* + *bnum* * *unit-length*, where

unit-length is the size of a single bucket

Step 8: access bucket D at *loc* and exit ■

C. Algorithm Insert

Input: file **F**, directory *dir* to **F**, and (K(L),S(L),C(L)),
where L ϵ **U**.

Output: updated file **F** such that **F** $\leftarrow$ **F** U {(K(L),S(L),C(L))}.

Step 1: access bucket D using Steps 2-8 of **Algorithm Access**

Step 2: read *db*, depth of bucket D

Step 3: #is D expandable bucket?#
if *db* > *maxd*, goto Step 7

Step 4: #here, D is fixed bucket#

a. if D has room for a new record,

1) insert (K(L),S(L),C(L)) to D
(if the key already exists, report error and stop)

2) for each element of the directory that points to D,
increment *occ* by 1 and stop

b. otherwise

1) obtain a new bucket D'

2) copy contents of D to *temporary area* and clear D

3) insert (K(L),S(L),C(L)) to *temporary area*

4) *db* $\leftarrow$ *db*+1 and set depths of D and D' with *db*

5) for each element of the directory that points to D,
set *occ* to 0

Step 5: #update directory#

a. if *db* $\leq$ *d*, #trivial case#

1) adjust the *ptr*s of directory that point to D such
that the lower indexed 2^{d-db} elements point to D
and the higher indexed 2^{d-db} elements point to D'

2) goto Step 6

b. if $db \leq maxd$, #needs directory doubling#

1) double the directory and increase d by 1

2) set $dir[2i+1].ptr$ to point to D' and goto Step 6

c. #here, linear split of bucket is needed#

1) set the split level, $dir[i].j$, to 1

2) allocate a new bucket (called *bucket*[1]) after bucket D (now, called *bucket*[0])

Step 6: #recursively insert records#

Insert each record in *temporary area* to **F** and exit

Step 7: #insert into expandable bucket#

a. if D is not full, insert the record into D

(if the key already exists, report error and stop)

b. otherwise,

1) insert the record into an overflow field of D

(if the key already exists, report error and stop)

2) update $dir[i].over$, if changed

Step 8: #update occupancy#

1) increase $dir[i].occ$ by 1

2) if load factor does not exceed upper limit, exit

Step 9: #here, load factor exceeds upper limit#

1) #update split level, if required#

if $dir[i].p = 0$, increase $dir[i].j$ by 1

2) allocate bucket[$p+2^{j-1}$], where p and j stand for $dir[i].p$ and $dir[i].j$, respectively

3) #split quadtree leaf node, if necessary#

if bucket[p] contains a single record,

replace it with four records, each corresponding to a quadrant of the original leaf node, and increase $dir[i].occ$ by 3

4) #reorganize records#
for each record in bucket[p],
if ($maxd+j$)th bit = "0", insert into bucket[p];
otherwise, insert into bucket[$p+2^{j-1}$]

5) update $dir[i].over$, if changed

6) $db \leftarrow db + 1$ and set depths of bucket[p] and bucket[$p+2^{j-1}$] with db

7) #update next bucket to split #
$dir[i].p \leftarrow (p+1) \bmod 2^{j-1}$

8) if load factor is within predefined range, exit;
otherwise, repeat Step 9 ■

Note:

When a file is initially constructed, a directory of depth 0 (with one element) is created and its j, p, occ and $over$ fields are set to zero. Also, a data bucket with depth 0 is created and its address is stored at the ptr field of the directory. Then, records are added into the file one by one using **Algorithm Insert.**

D. Algorithm Delete

Input: file **F**, directory *dir* to **F**, and K(L), where L ϵ **U**.

Output: updated file **F** such that **F** $\leftarrow$ **F** - {(K(L),S(L),C(L))}.

Step 1: access bucket D using Steps 2-8 of **Algorithm Access**

Step 2: read *db*, depth of bucket D

Step 3: #is D expandable bucket?#

if *db* > *maxd*, goto Step 6

Step 4: #here, D is fixed bucket#

1) delete (K(L),S(L),C(L)) from D

(if not found, report error and stop)

2) for each element of the directory that points to D,

decrease *occ* by 1

Step 5: #terminate?#

a. if D is more than half full, exit

b. if buddy of D(say, D') is more than half full, exit

c. #here, bucket merge is triggered#

1) merge D' into D and free D'

2) adjust each element of the directory pointing to D'

to point to D

3) for each element of the directory pointing to D

update *occ*

4) decrease depth of bucket D by 1

5) if depth of D is 0, stop; otherwise, repeat Step 5

Step 6: #delete from expandable bucket#

1) delete (K(L),S(L),C(L)) from D or from an over-

flow field of D (if not found, report error and

stop)

2) update $dir[i].over$, if changed

3) decrease $dir[i].occ$ by 1

Step 7: #terminate?#

if load factor is within predefined range, exit

Step 8: #here, load factor falls below lower limit#

a. if $dir[i].j = 1$, #split level is 1#

1) merge bucket[1] to bucket[0] and free bucket[1]

2) if bucket[0] contains mergeable records, merge them into one and decrease $dir[i].occ$ by 3

3) update $dir[i].over$, if changed

4) decrease depth of bucket[0] by 1

5) set $dir[i].j$ to 0

6) let bucket[0] be D and goto Step 5

b. if $dir[i].p = 0$, #every bucket has been split#

1) set $dir[i].p \leftarrow 2^{j-1}-1$

(hereafter, p and j may be used to denote $dir[i].p$ and $dir[i].j$, respectively)

2) merge bucket[$p+2^{j-1}$] to bucket[p] and free bucket[$p+2^{j-1}$]

3) if bucket[p] contains mergeable records, merge them into one and decrease $dir[i].occ$ by 3

4) update $dir[i].over$, if changed

5) decrease depth of bucket[p] by 1 and goto Step 7

c. #here, some buckets have been split, others not#

1) decrease $dir[i].p$ by 1

2) merge bucket[$p+2^{j-1}$] to bucket[p] and free

bucket[p+2^{j-1}]

3) if bucket[p] contains mergeable records, merge them into one and decrease $dir[i].occ$ by 3

4) update $dir[i].over$, if changed

5) if $dir[i].p$ = 0, decrease $dir[i].j$ by 1

6) decrease depth of bucket[p] by 1 and goto Step 7 ■

Note:

The criteria for merging fixed data buckets are to be determined by the implementer and do not have to follow the conditions used in Steps 5a and 5b.

E. Algorithm SequenRetrieve

Input: file **F** and directory *dir* to **F**.

Output: each data bucket in **F** retrieved in order such that sequential processing of records is possible without accessing a same data bucket more than once.

Step 1: read *d*, depth of directory

Step 2: #initialize variables#

i ← 0 and *loc* ← null

Step 3: #determine address of next bucket#

a. if *dir*[*i*].*ptr* ≠ *loc*, *loc* ← *dir*[*i*].*ptr*;

b. otherwise, #skip the duplicated element#

goto Step 7

Step 4: #fixed bucket? #

if *dir*[*i*].*j* = 0,

retrieve bucket at *loc* and goto Step 7

Step 5: #compute *M* (no. of expandable buckets)#

if *dir*[*i*].*p* = 0, $M \leftarrow 2^j$; otherwise, $M \leftarrow 2^{j-1}+p$

Step 6: #retrieve *M* buckets in order#

1) initialize subblock index *isub* to 0

2) #calculate bucket no. #

bnum ← $\Sigma a_k 2^{k-1-k}$, for $0 \leq k \leq m-1$, where

isub is equivalent to $\Sigma a_k 2^k$, for $0 \leq k \leq m-1$

3) if *bnum* ≥ *M*, goto Step 6.7

4) #calculate address of the bucket#

addr ← *loc* + *bnum* * *unit-length*, where

unit-length is the size of a single bucket

5) retrieve bucket D at *addr*

6) retrieve overflow fields attached to D, if any

7) increase *isub* by 1

8) if *isub* < 2^j, repeat Step 6

Step 7: #continue until the end of the directory#

1) increase *i* by 1

2) if *i* = 2^d, stop; otherwise, goto Step 3 ■

F. Algorithm RangeSearch

Input: file **F**, directory *dir* to **F**, and (x1,y1),(x2,y2) ϵ **U**, where x1 $\leq$ x2 and y1 $\leq$ y2.

Output: retrieve every data bucket whose associated block or subblock in **U** intersects the query rectangle specified by (x1,y1) and (x2,y2).

Step 1: #initialize a set to store the indices#

R $\leftarrow$ { }

Step 2: read *d*, depth of directory

Step 3: #calculate keys and corresponding indices#

key1 $\leftarrow$ *s*(x1,y1); *i1* $\leftarrow$ $\lfloor key1 \,/\, 2^{2n-d} \rfloor$

key2 $\leftarrow$ *s*(x2,y2); *i2* $\leftarrow$ $\lfloor key2 \,/\, 2^{2n-d} \rfloor$

Step 4: if *i1* = *i2*, #trivial case#

R $\leftarrow$ {*i1*} and goto Step 12

Step 5: #set up upper and lower limits of indices#

a. if *d* is even,

1) $xlow \leftarrow \Sigma a_{2k+1}2^k$, for $0 \leq k \leq \lfloor m/2 \rfloor$

2) $xhigh \leftarrow \Sigma b_{2k+1}2^k$, for $0 \leq k \leq \lfloor m/2 \rfloor$

3) $ylow \leftarrow \Sigma a_{2k}2^k$, for $0 \leq k \leq \lceil m/2 \rceil$

4) $yhigh \leftarrow \Sigma b_{2k}2^k$, for $0 \leq k \leq \lceil m/2 \rceil$

b. otherwise,

1) $xlow \leftarrow \Sigma a_{2k}2^k$, for $0 \leq k \leq \lceil m/2 \rceil$

2) $xhigh \leftarrow \Sigma b_{2k}2^k$, for $0 \leq k \leq \lceil m/2 \rceil$

3) $ylow \leftarrow \Sigma a_{2k+1}2^k$, for $0 \leq k \leq \lfloor m/2 \rfloor$

4) $yhigh \leftarrow \Sigma b_{2k+1}2^k$, for $0 \leq k \leq \lfloor m/2 \rfloor$

where *i1* (*i2*) is equivalent to $\Sigma a_k 2^k$ ($\Sigma b_k 2^k$),

for $0 \le k \le m-1$

Step 6: #initialize *x* index#

ix ← *xlow*

Step 7: #initialize *y* index#

iy ← *ylow*

Step 8: #compute the index of relevant block#

if *d* is even, *i* ← **shuffle**(*ix*,*iy*);

otherwise, *i* ← **shuffle**(*iy*,*ix*),

where **shuffle** is a function defined as follows:

Given $V = \Sigma v_k 2^k$ and $W = \Sigma w_k 2^k$,

shuffle$(V,W) = \Sigma(2v_k+w_k)2^{2k}$, for $0 \le k \le m-1$

Step 9: #store the index#

$R \leftarrow R \cup \{i\}$

Step 10: #continue until y upper limit is reached#

1) increase *iy* by 1

2) if *iy* ≤ *yhigh*, goto Step 8

Step 11: #continue until x upper limit is reached#

1) increase *ix* by 1

2) if *ix* ≤ *xhigh*, goto Step 7

Step 12: #initialize a variable#

loc ← null

Step 13: for each member *i* in R, perform the following:

a. #determine address of bucket#

if *dir*[*i*].*ptr* ≠ *loc*, *loc* ← *dir*[*i*].*ptr*;

otherwise, #skip the duplicated element#

goto Step 13.e

b. if *dir*[*i*].*j* = 0, #fixed bucket#

retrieve bucket at *loc* and goto Step 13.e

c. #here, linear splits have occurred#
   if the block is totally contained in the query rectangle, retrieve every bucket belonging to the block using Steps 5-6 of **Algorithm SequenRetrieve** and goto Step 13.e

d. #the block is partially contained#

1) #initialize a set to store *isub*, subblock indices#
   R' ← { }

2) let (x1',y1') and (x2',y2'), where x1' ≤ x2' and y1' ≤ y2', be the points specifying the rectangle which is the intersection of the current minimal block and the query rectangle

3) compute R' in a similar manner to Steps 3-11
   Note: in Step 8, *d* should be substituted with *j*

4) #compute the number of buckets#
   if $dir[i].p = 0$, $M \leftarrow 2^j$; otherwise, $M \leftarrow 2^{j-1}+p$

5) for each member *isub* in R',
      compute bucket number *bnum*;
      if *bnum* > *M*,
         calculate address of bucket and retrieve

e. continue ■

Note:

If a query rectangle is not completely contained in **U**, it should be modified before **Algorthm RangeSearch** is called such that the modified rectangle is the intersection of **U** and the original query rectangle.

University of Alberta Library
0 1620 0134 0676
B34339